NEC SX-Aurora TSUBASA ではじめる
ベクトルプログラミング

滝沢 寛之

はじめに

NEC SX-Aurora TSUBASA は 2018 年 2 月に発売された最新のベクトル型計算システムです。本文書の目的は、SX-Aurora TSUBASA の基本構成や重要な概念を説明し、SX-Aurora TSUBASA 向けに性能を意識したプログラミング (**高性能計算プログラミング**) を行う際の指針を示すことです。

SX-Aurora TSUBASA の特筆すべき特長として、**ベクトルアーキテクチャ**の特別なハードウェア構成を採用しながら、標準的なソフトウェア環境を利用可能であるという点が挙げられます。C/C++ や Fortran といった一般的なプログラミング言語を使ってプログラムを開発し、そのプログラムを一般的な PC と同様の Linux OS 上で実行することができます。これは従来のベクトルシステムとは大きく異なる特長であり、SX-Aurora TSUBASA が期待される大きな理由の一つとなっています。

ベクトルアーキテクチャとしての性質上、SX-Aurora TSUBASA の性能を引き出すための最も重要な秘訣は、**ベクトル命令**をできるだけ多く使って計算することです。ベクトル命令とは、そのオペランド (処理対象) がベクトルになっている命令です。数学で出てくるベクトルと同様に、複数の数値 (ベクトル要素) をまとめて 1 つのベクトルとして扱う命令がベクトル命令だと言えます。複数のベクトル要素に対する演算を並列実行し、さらにその演算自体もパイプライン実行することによって、ベクトルアーキテクチャは高性能を実現しています。現在、一般的な PC に搭載されているプロセッサですら一種のベクトル命令 *[1]を持っていますが、SX-Aurora TSUBASA のベクトル命令は 256 要素もの長いベクトルを処理することができるという点で、他のプロセッサのベクトル命令とは一線を画しています。現在、SX-Aurora TSUBASA は現存する唯一の真のベクトルアーキテクチャであり、高性能計算分野で独自の地位を確立しています。

ベクトル命令をできるだけ多く使うことが最重要ではありますが、プログラマが機械語命令 (あるいは、アセンブリコード) を直接書くのは現実的ではありません。ほとんどの場合、プログラマは C/C++ や Fortran といった高級言語でプログラムを記述し、それをコンパイラが機械語命令へと翻訳 (コンパイル) します。その際に、コンパイラはベクトル命令で実行可能な部分を見つけて、実際にその部分がベクトル命令で実行されるようにプログラムを生成する機能があります。これをコンパイラによる自動**ベクトル化**機能と呼びます。コンパイラによる自動ベクトル化を最大限に活用することが、SX-Aurora TSUBASA の計算能力を引き出すために極めて重要になります。このためには、コンパイラの事情にあわせて、コンパイラが自動ベクトル化しやすいように配慮してプログラムを書く必要があります。本文書では、コンパイラによるベクトル化を意識したプログラミングを**ベクトルプログラミング**と呼び、SX-Aurora TSUBASA 向けの高性能計算プログラミングの中でも特に重要だと考えて詳しく説明します。

本文書では、上述の新しい世代のベクトルアーキテクチャである SX-Aurora TSUBASA を活用するためのプログラミングについて説明します。本文書は 2 部構成になっています。第 1 章から第 5 章までは第 1 部 基礎編であり、SX-Aurora TSUBASA の基本的な使い方や、性能を出すために考えなくてはならない基礎的な知識について説明します。第 6 章以降は第 2 部 応用編であり、さらに性能を高めるための高度な話題や、高性能計算プログラミングの具体例を示します。

第 1 章では、SX-Aurora TSUBASA アーキテクチャの概要を説明し、ハードウェア構成やその性能面からの特徴を

*[1] Single Instruction Multiple Data (SIMD) 命令と呼ばれています。

紹介します。ベクトルアーキテクチャとしての特徴から、どのような場合に SX-Aurora TSUBASA は高性能を実現できるのかを説明します。また、SX-Aurora TSUBASA はベクトルアーキテクチャの特別なハードウェア構成を採用しながら、標準的なソフトウェア環境で利用できる仕組みについても説明します。

第 2 章では、SX-Aurora TSUBASA 向けにプログラムを書いて実行するまでの一連の手順を説明します。SX-Aurora TSUBASA のプログラミングには、C/C++ や Fortran と言った一般的なプログラミング言語を使うことができます。SX-Aurora TSUBASA 向けに提供されている、プログラミング環境の使い方について説明します。

第 3 章では、プログラムの性能解析を行う方法を説明します。高性能プログラミングにおいて、プログラムのどの部分で時間を費やしているのか、期待どおりの性能が出ているのか、などを解析することは非常に重要です。第 3 章では、実際に利用可能な関連コマンドを紹介しながら、SX-Aurora TSUBASA の性能を解析する方法を説明します。

第 4 章では、ベクトル化を促進するためのいくつかの要点を説明します。まず、コンパイラがプログラムをどのようにコンパイルしたのかを確認する方法を説明します。次に、コンパイラがループをベクトル化できると判断するための条件を紹介し、その条件を満足するようにループ構造を修正する方法等を紹介します。また、ベクトル長を調整することでベクトル演算の効率を高める方法についても述べます。

第 5 章では、SX-Aurora TSUBASA のメモリアクセスを最適化する方法を説明します。SX-Aurora TSUBASA のメモリアクセス性能は、データへのアクセス順序や方法によって大きく変化します。このため、どのようなアクセスをすれば高いメモリアクセス性能が実現できるのかを説明し、性能を高めるための考え方や要点を示します。

第 6 章では、SX-Aurora TSUBASA 独自の Vector Host Call (VH Call) や Vector Engine Offloading (VEO) の使い方について説明します。

第 7 章では、実アプリ最適化事例を紹介します。具体的には、プログラム性能解析から始まって、計算の順番を入れ替えることでベクトル化を促進することを考え、その効果を定量的に確認します。

第 8 章では、以上の章では言及されなかったその他の話題について紹介します。

目次

第 I 部　基礎編 7

第 1 章　アーキテクチャの概要 9
1.1 背景 . . . 9
1.2 SX-Aurora TSUBASA アーキテクチャ . . . 11

第 2 章　プログラミング環境の概要 19
2.1 システム構成の確認 . . . 19
2.2 プログラムのコンパイルと実行 . . . 22
2.3 プログラムのデバッグ . . . 27

第 3 章　性能解析 33
3.1 性能指標 . . . 33
3.2 プログラム全体の性能情報 . . . 35
3.3 詳細な性能情報 . . . 38

第 4 章　ベクトルプログラミング 43
4.1 ループのベクトル化 . . . 43
4.2 コンパイルリスト . . . 45
4.3 ベクトル化の促進 . . . 49
4.4 ベクトル長の増加 . . . 58

第 5 章　メモリアクセス最適化 63
5.1 メモリシステムの構成 . . . 63
5.2 メモリアクセス最適化 . . . 67
5.3 メモリ階層の効果的利用 . . . 73

第 II 部　応用編 75

第 6 章　異種プロセッサ間の連携 77

第 7 章　プログラム開発事例 79

第 8 章　その他の高度な話題 81

付録 A　OpenMP 超入門 83

参考文献 87

索引 88

第 I 部

基礎編

第 1 章

アーキテクチャの概要

コンピュータの設計者は、様々な制約のもとでコンピュータを設計しています。できることなら、どんなプログラムでも速く実行するような万能のコンピュータを作りたいのはやまやまですが、そのようなコンピュータは許容できないくらい高価になったり、様々な物理制約 (微細化の限界、消費電力や発熱など) からそもそも作れなかったりします。このため、コンピュータ設計では

Make the common case fast (一般的な場合を高速化する) [6]

つまり、よくある (頻繁に実行される) 一般的な処理が速くなるように設計することが、重要な基本方針の一つになっています。よくある処理を高速化できれば、その処理を行うたびに高速化の恩恵を受けられます。逆に、滅多に実行されない処理を高速化できても、その高速化の恩恵を受けられる機会は稀になってしまいます。仮に同じコストをかけて高速化するとしたら、前者の処理を高速化したほうが効果的であることは容易に想像できることでしょう。「頻繁に実行される重要な処理」だとコンピュータ設計者が考える処理が優先的に高速化された結果、現代のコンピュータでは処理の得手不得手がますます顕著になっています。

高速に動作するプログラムを開発するためには、対象となるシステムがどのような処理を得意としているのかを知ることが重要です。本章では、SX-Aurora TSUBASA のアーキテクチャの概要を説明し、どのような処理が common case として想定されているのかを説明します。

1.1 背景

まず、SX-Aurora TSUBASA が登場した背景について簡単に説明します。

動作周波数の向上が難しくなって以降、プロセッサの理論演算性能[*1]のさらなる向上のために採れる手段はあまり多くありません。図 1.1 に、Intel 社のハイエンドプロセッサの年ごとの性能向上を示します。この図から、大まかには 2 つの手段を組み合わせてプロセッサの性能を高めてきたことがわかります。1 つはプロセッサに搭載するコアの数を増やすことで、もう 1 つはベクトル命令 (SIMD 命令) の長さを増やすことです。1998 年までは、プロセッサの性能向上は動作周波数と連動していました。しかし、1999 年には単精度浮動小数点演算のベクトル命令 (Streaming SIMD Extension, SSE) が導入され、単精度浮動小数点演算性能と動作周波数の向上率が乖離しました。その後、2001 年には倍精度浮動小数点演算のベクトル命令 (Streaming SIMD Extension 2, SSE2) も導入され、動作周波数と浮動小数点演算性能の向上率は一致しなくなりました。さらに、2005 年にはプロセッサ (ソケット) が複数のコアを持つようになり、プロセッサの性能とコアあたりの性能も一致しなくなりました。それ以降も動作周波数はほとんど向上していませんが、プロセッサあたりの演算性能およびコアあたりの演算性能は向上し続けています。後者は主にベクトル長の増

[*1] プロセッサが達成できる演算性能の理論的な上限値。

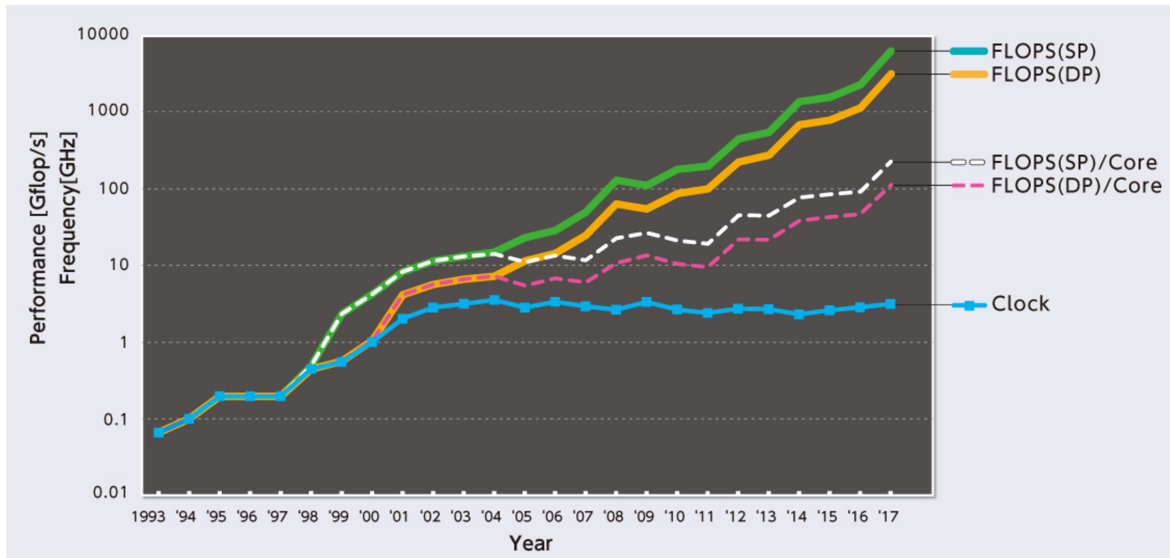


図 1.1　プロセッサの性能向上

加、前者はさらにコア数も増加させることにより実現されています。

図 1.1 の Intel 社製のプロセッサはかつて**スカラプロセッサ**と表現されることも多かったのですが、現在ではスカラプロセッサという用語自体が死語になっているように思います。これは、スカラプロセッサの対義語である (古典的な意味での) ベクトルプロセッサが今となってはほとんどなくなったという理由もありますが、すべてのプロセッサがベクトルプロセッサになったためにスカラプロセッサという表現が実態に合わなくなったのも大きな理由だと思われます。例えば最新 (2018 年時点) の Skylake 世代の Intel 社製プロセッサでは、AVX-512 命令という一種のベクトル命令によって、8 要素のベクトルの倍精度積和演算を行うことで高性能を実現しています。現在利用可能なほぼすべてのプロセッサにおいて、その演算性能を引き出すためにはベクトル命令の活用が欠かせません。

それでは、すべてのプロセッサがベクトルプロセッサになった現在、SX-Aurora TSUBASA の特長はどこにあるのでしょうか？その謎は図 1.1 を見ているだけでは解けません。まずは図 1.2 を見てください。これは図 1.1 に示したプロセッサあたりの演算性能に加えて、メモリバンド幅の向上を示したものです。縦軸を線形にしているため、演算性能が指数関数的に順調に伸びていることがわかりやすいと思いますが、その一方でメモリバンド幅の向上率がかなり低いことも明確にわかります。コンピュータはメモリ上のデータを読み込んで何らかの処理を行い、その結果をまたメモリに書き戻すことで動作しているため、図 1.2 に示すように性能差が大きく広がってしまうと、例え理論演算性能が向上しても、メモリバンド幅が足りずに実際の性能 (**実効性能**) を高めることはできません。もちろん、現代のプロセッサではメモリバンド幅不足を補うためにキャッシュメモリなどの様々な工夫が用いられていますが、それでもなお、高いメモリバンド幅が求められるアプリケーションは特に科学技術計算の分野で非常に多く見られます [8]。このため、高いメモリバンド幅を実現し、科学技術計算で高い実効性能を達成できるシステムの登場が強く求められてきました。そのニーズに応えるため、科学技術計算に主眼をおいて設計された SX-Aurora TSUBASA は、高いメモリバンド幅という非常に重要な特長を持っています。

実は、SX-Aurora TSUBASA に限らず、従来の NEC SX シリーズも同世代のプロセッサと比較して非常に高い**理論メモリバンド幅**を実現してきました。図 1.2 とは異なり、見やすさのために縦軸を対数にして、年ごとの演算性能とメモリバンド幅の向上の様子を図 1.3 に示します。高い理論メモリバンド幅は SX-Aurora TSUBASA の特長というよりも NEC SX シリーズやそのベクトルアーキテクチャの持つ特長と言えます。しかしながら、従来の SX と最新の SX-Aurora TSUBASA とは決定的に違う点があります。それは、SX-Aurora TSUBASA よりも前の世代の SX ではソフトウェア環境が特殊であったという点です。例えば、NEC SUPER-UX という専用 OS の環境で使う必要があり

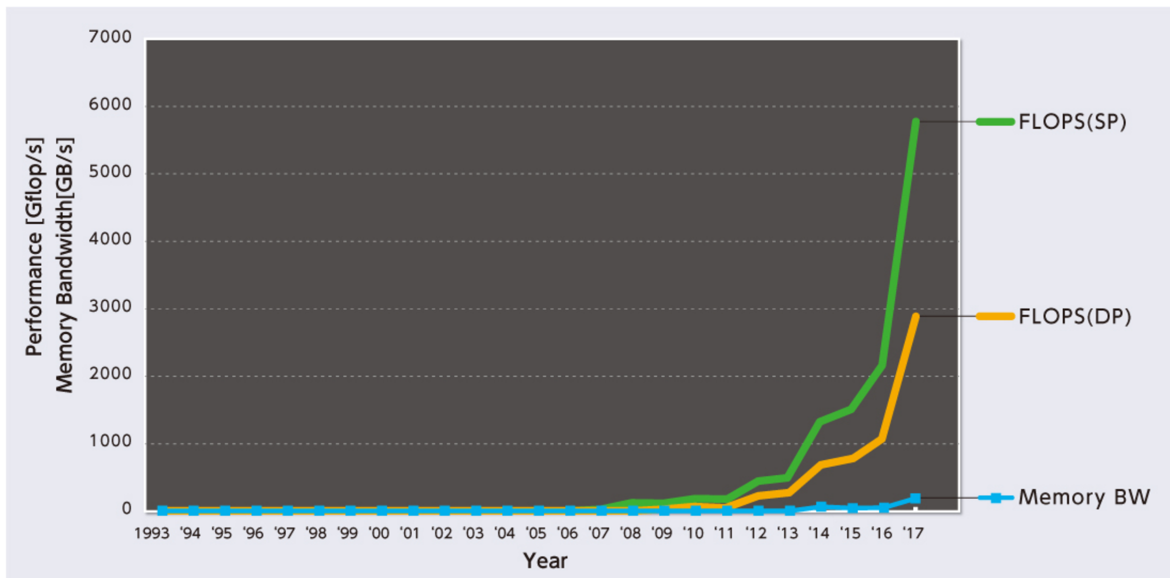


図 1.2 理論演算性能と理論メモリバンド幅の向上

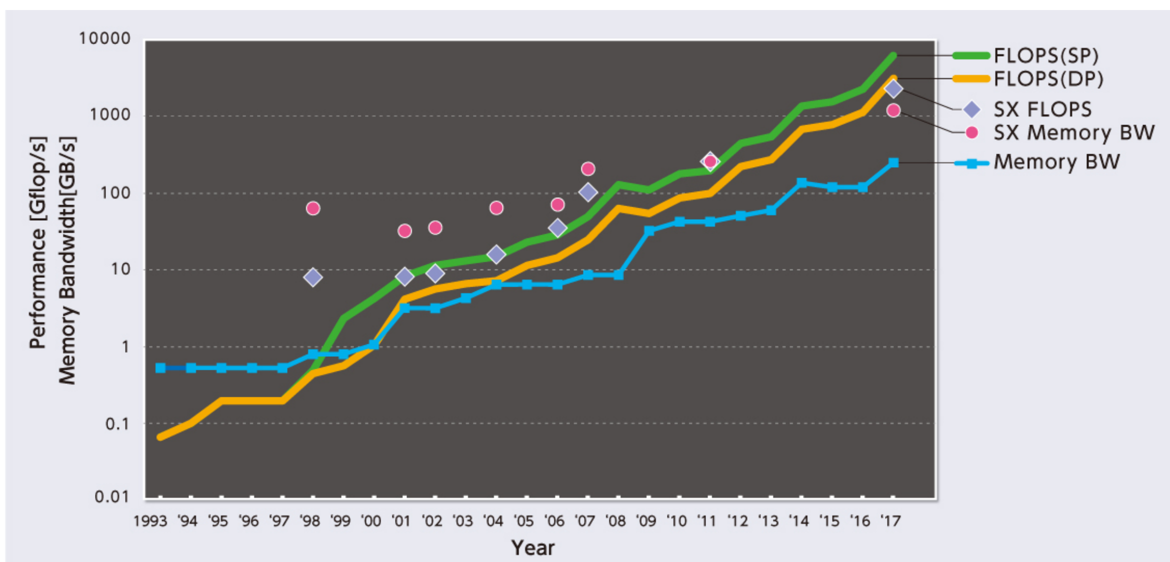


図 1.3 SX シリーズの演算能力とメモリバンド幅の向上

ました。このため、標準的な環境向けに開発されていきたソフトウェアの多くが、従来の SX のソフトウェア環境ではそのままでは使えないという問題がありました。一方、SX-Aurora TSUBASA は科学技術計算用システムで広く使われている Linux OS の管理下で動作します。すなわち SX-Aurora TSUBASA は、高いメモリバンド幅というハードウェア面での特長と、標準環境で利用可能になったというソフトウェア面での特長を併せ持っています。

1.2 SX-Aurora TSUBASA アーキテクチャ

SX-Aurora TSUBASA のシステム構成を図 1.4 に示します。SX-Aurora TSUBASA では、一般的な x86 プロセッサを制御用に使うことが想定されており、ベクトルプロセッサが演算に専念できる構成になっています。SX-Aurora TSUBASA では制御用プロセッサをベクトルホスト (Vector Host, VH)、演算用のベクトルプロセッサをベクトルエ

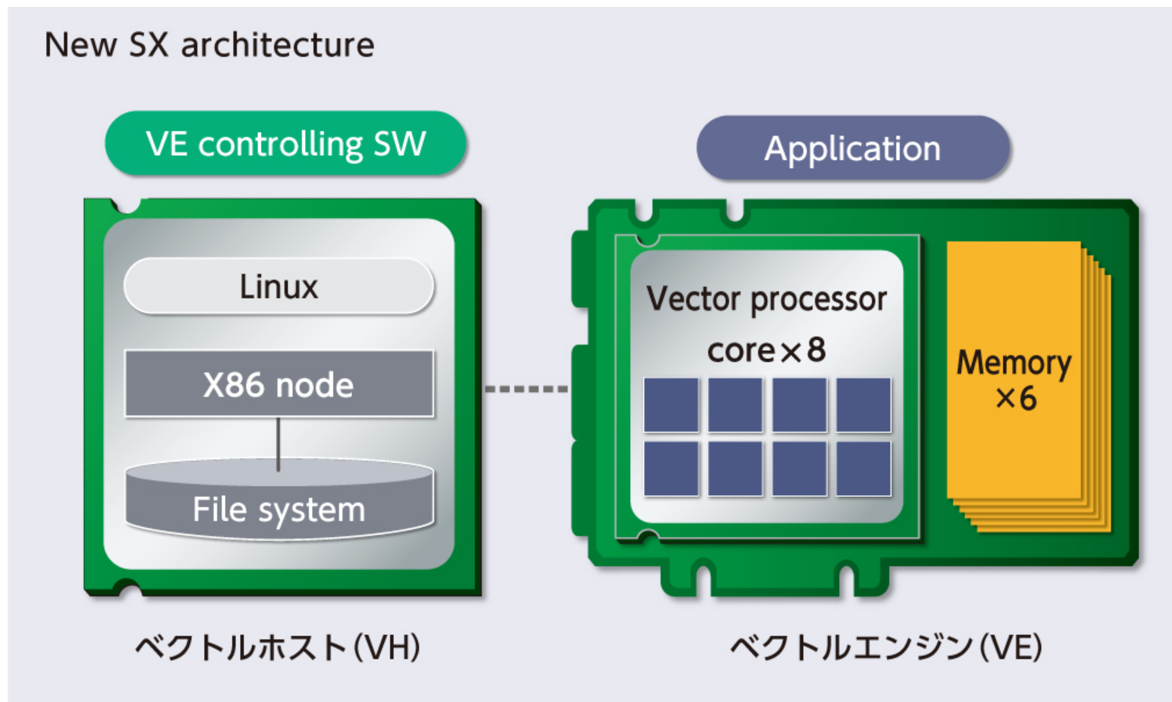


図 1.4　SX-Aurora TSUBASA のシステム構成

ンジン (Vector Engine, VE) と呼びます。本節では、まず最初に VE のアーキテクチャについて説明し、その後に VH と VE とを連携させるソフトウェア環境について説明します。

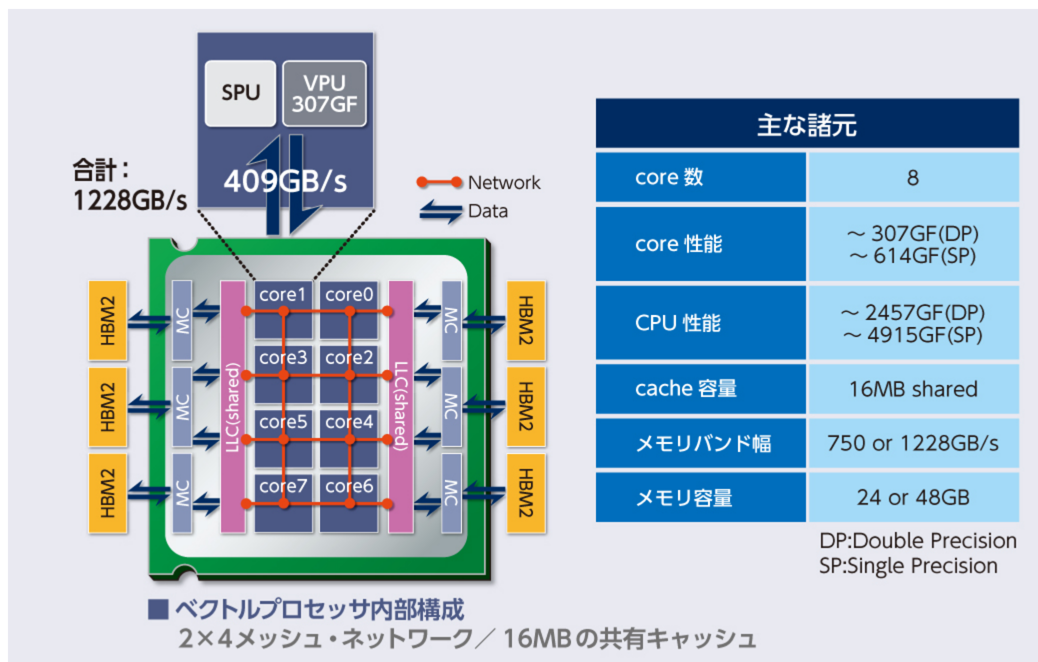
1.2.1　ベクトルエンジン

VE はメモリバンド幅を重視した設計になっており、それによって高速に読み込んだ大量のデータを一括処理する長いベクトル命令が効果を発揮します。これが SX-Aurora TSUBASA の得意とする処理、すなわち本章の冒頭で説明した「common case」です。図 1.3 から、プロセッサあたりのメモリバンド幅という観点では、VE が通常のプロセッサに対して圧倒的な優位にあることがわかります。この性能特性を活かすことが、SX-Aurora TSUBASA を活用した高性能計算の実現であると言い換えることができます。

VE の高いバンド幅は、高速なメモリデバイスをプロセッサチップにより多く接続することで実現されています。VE のプロセッサチップの写真と内部構成を図 1.5 および図 1.6 に示します。VE プロセッサは 8 コアのマルチコア構成になっており、その周りに第 2 世代の *High Bandwidth Memory* (*HBM2*) と呼ばれるメモリチップが 6 つ並んでいます。HBM2 の規格として、1 つのチップ (モジュール) あたり最大で 256 GB/s という非常に高いバンド幅を実現できる仕様になっています。SX-Aurora TSUBASA の VE では、HBM2 を 204.6 GB/s で動作させて、6 モジュールに並列アクセスすることで $204.6 \times 6 = 1228$ GB/s のメモリバンド幅を達成しています[*2]。これは 2019 年現在、プロセッサあたりのメモリバンド幅としては世界最高値です。ただし、図 1.7 に示すとおり、メモリとベクトルレジスタの間にはキャッシュが搭載されており、メモリとキャッシュ間は 1228 GB/s の帯域が確保されていますが、各コアとキャッシュ間の帯域は 409.2 GB/s になっています。コアとキャッシュの間のネットワークの総バンド幅は 3072 GB/s です。このため、キャッシュ上のデータに 8 コアすべてでアクセスした場合には 3072 GB/s の高い総バンド幅でアクセスできますが、コアあたりのバンド幅はデータがキャッシュ上にあってもメモリ上にあっても最大で 409.2 GB/s というこ

[*2] 廉価版 VE の Type 10C では $125 \times 6 = 750$ GB/s

図 1.5　VE のプロセッサチップの写真



主な諸元	
core 数	8
core 性能	～307GF(DP) ～614GF(SP)
CPU 性能	～2457GF(DP) ～4915GF(SP)
cache 容量	16MB shared
メモリバンド幅	750 or 1228GB/s
メモリ容量	24 or 48GB

図 1.6　VE のプロセッサ内部構成

となります。したがって、複数のコアを使って並列計算を行うということが、演算性能という観点からだけではなくて実際に達成されるメモリバンド幅 (**実効メモリバンド幅**) という観点からも重要になります。

高いメモリバンド幅を活用するために、VE は最大 256 要素を一括して処理できるベクトル命令を持っています。256 要素のデータを保持できる**ベクトルレジスタ**を、ベクトル命令のオペランドとして指定します。ベクトル命令には、大きく分けて 256 要素の倍精度浮動小数点データを 1 命令で読み書きする**ベクトルロード/ストア命令**や、256 要素の演算を 1 命令で行う**ベクトル演算命令**があります。ベクトルロード/ストア命令では、メモリとベクトルレジスタとの間のデータ転送を最大で 256 要素単位で指示するため命令です。倍精度浮動小数点データは 1 要素で 8 バイトですの

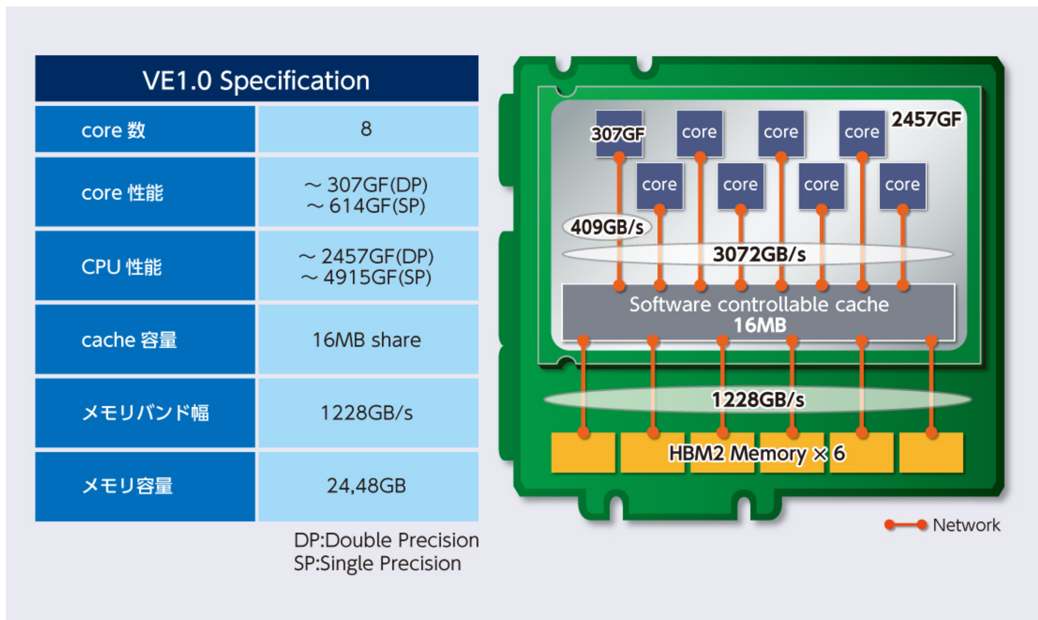


図 1.7 メモリバンド幅とキャッシュバンド幅

で、256 要素では 2 KB ものデータ転送を 1 命令で指示できることになります。一方、ベクトル演算命令はベクトル間の演算を指示します。VE は、32 個の倍精度浮動小数点データの**融合積和演算** (fused multiply-add, *FMA*) を同時実行できる FMA 演算器 (ベクトル FMA 演算器) が搭載されています。それぞれの演算には 8 サイクルかかるのですが、図 1.8 に示すように、ベクトル FMA 演算器はパイプライン的に動作して各サイクルに 32 積和演算 (64 浮動小数点演算, 64 flop) を同時実行できるので、十分に長いベクトル処理においては 64 flop/cycle× 1.6 GHz = 102.4 Gflop/s の浮動小数点演算性能となります。そのようなベクトル FMA 演算器が 1 コアあたり 3 基搭載されているので、1 コアの浮動小数点演算性能は 307.2 Gflop/s です。図 1.1 に示すとおり、最新の Intel プロセッサのコアあたりの性能が約 100 Gflop/s ですので、VE のコアあたりの演算性能が高いことがわかります。なお、1 つの VE には 8 コアが搭載されているので、VE の浮動小数点演算性能は $8 \times 307.2 = 2457.6$ Gflop/s となります。以上では倍精度浮動小数点演算の性能値を計算してきましたが、単精度の場合には各サイクルに 64 積和演算を実行できるため、それぞれの性能値は 2 倍になります。コアあたりの単精度浮動小数点演算性能は 614.4 Gflop/s で、プロセッサあたりでは 614.4 Gflop/s × 8 cores = 4915.2 Gflop/s です。

以上のハードウェア構成から、VE は大量のデータに対してベクトル演算を行う処理を得意とすることがわかるでしょう。VE にはベクトル命令以外の命令を処理するための**スカラユニット**も搭載されていますが、その演算性能はベクトル演算ユニットと比べるとかなり低いものとなってます。したがって、なるべく多くの演算をベクトル命令を使って行うことが非常に重要です。また、3 コア以上を使ってメモリアクセスしなければ 1228 GB/s のバンド幅を有効活用できないので、並列プログラミングも重要です。さらに、データの再利用性が高い場合、全コアの合計で 3072 GB/s ものバンド幅でキャッシュ上のデータにアクセスすることができますので、キャッシュに配慮したプログラミングも求められます。これらに配慮したプログラミングを本文書では解説していきます。

1.2.2 ソフトウェア環境

図 1.4 に示した通り、SX-Aurora TSUBASA のシステムは VH と VE から構成されており、VH と VE は物理的には切り離されています。しかし、VH でプログラムを実行するのと似たような感覚で、VE のプログラムを実行することができます。SX-Aurora TSUBASA では、VH が標準的な Linux OS で動作しているので、あたかも Linux のプロ

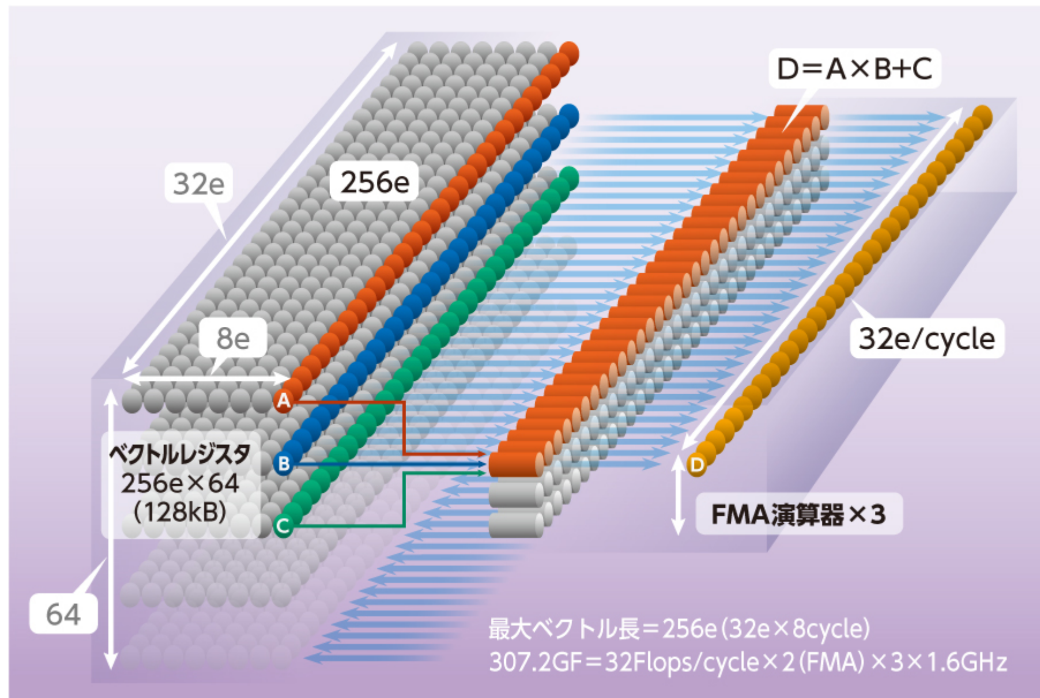


図 1.8 ベクトル演算命令の動作イメージ

グラムを実行しているような感覚で VE の演算性能を活用することができます (詳しくは第 2 章参照)。第 1.2.2 節では、そのような使い勝手を実現しているソフトウェア環境の概要について説明します。

Linux などの一般的な OS では、(OS の) **カーネル**と呼ばれる中核部分が様々なシステム構成要素を計算資源 (**リソース**) として抽象化し、ハードウェアの機能をシステム利用者 (ユーザ) が起動したプログラムから間接的に利用できるようにしています。例えば、CPU 時間やメモリ領域が典型的なリソースです。ユーザのプログラムを起動したとき、カーネルはそのプログラム用のリソースを確保して**プロセス**という単位で管理します。ユーザのプログラムは、カーネルから割り当てられたリソースの範囲内で動作することになります。メモリ領域を例として考え、一般的な x86 システムで各プロセスに固有のメモリ領域が割り当てられている様子を図 1.9 に示します。この状態で、プロセスが範囲外のメモリ領域にアクセスしようとした場合にはセグメンテーション違反 (segmentation fault) として検出され、実際には範囲外へのアクセスが出来ないようになっています。その結果、同一システム上に複数のプロセスが動いている場合に、お互いに他のプロセスへ意図しない影響を与えないようになっています。物理的には同じメモリを使っていても、論理的には分割されており、プロセス間に明確な区切りがあります。

また図 1.9 に示すとおり、ユーザプロセスとカーネルの間にも明確な区切りがあります。両者は別のメモリ空間で動作しており、それぞれ**ユーザ空間**および**カーネル空間**と呼ばれています。一般的なシステム構成では、プロセスとカーネルは物理的には同じメモリを使って動作していることが多いのですが、論理的には別々のメモリ領域で動作しています。ユーザプロセスとカーネルが連携するためには、必要なデータを明示的に受け渡しする必要があります。プロセスとカーネルが通信するための仕組みは、**システムコール** (system call) と呼ばれます。プロセスが OS の機能を必要とする時には、システムコールによってカーネルに動作を要求します。カーネルは普段は待機状態になっているのですが、システムコールによって起動され、OS の中核としての仕事を行います。その結果をプロセスに返した後に、再び待機状態になります。

以上のように OS の基本的な構成を考えると、あるプロセスが物理的に切り離された別のプロセッサやメモリを使って動作していても、Linux OS で他のプロセスと同様に管理できそうだということがわかります。図 1.9 の下部に示

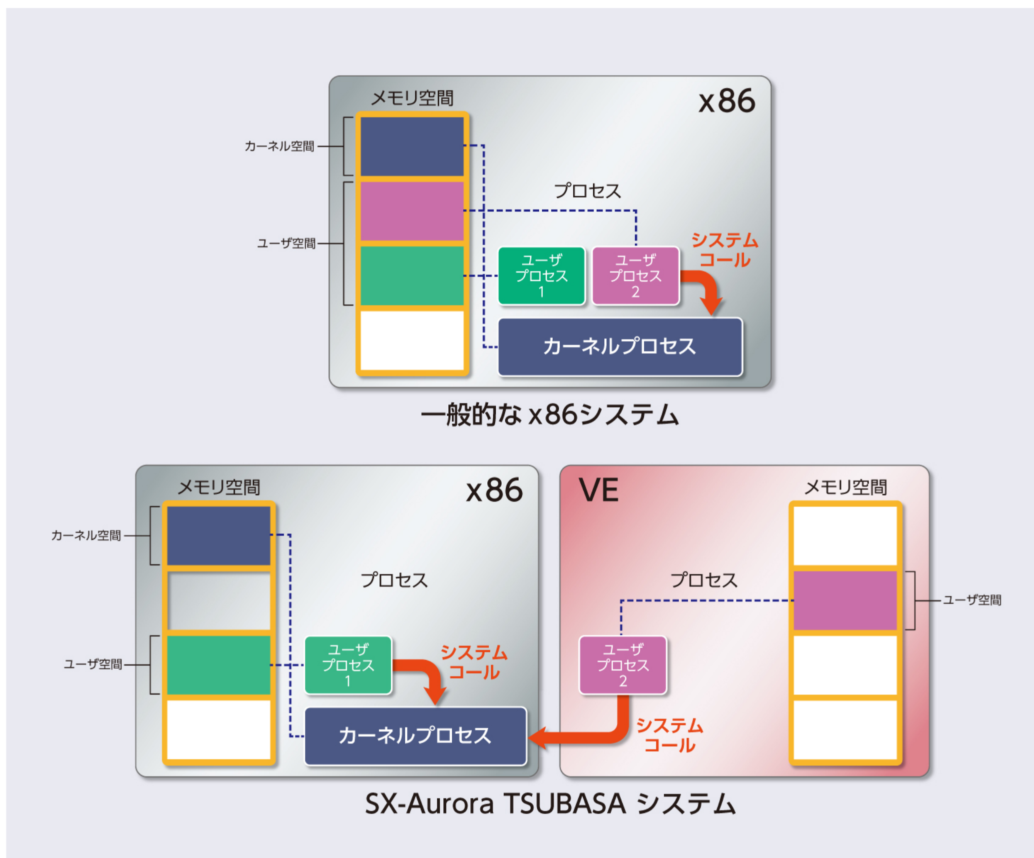


図 1.9　カーネルとプロセス

すとおり、SX-Aurora TSUBASA はまさにそのようなプロセス管理方式を採用しています。第 2 章で説明するように、VE 用に作成したプログラムを VE で実行するように指示することで、起動されたプロセスは物理的には VE 上で動作します。カーネル空間とユーザ空間は論理的にはもともと別の空間として管理されていたのですが、SX-Aurora TSUBASA ではそれらが物理的にも切り離されています。つまり、カーネル空間は常に VH 側のメモリに確保されますが、VE で動作しているプロセスのメモリ空間は物理的には VE 側のメモリに確保されます。そのようなプロセスがシステムコールを行った場合には、VE と VH との間で通信することで通常の Linux のプロセスと同様に OS の機能を利用することができます [9]。このようにして、SX-Aurora TSUBASA は Linux という標準環境でプログラムを開発・実行できるというソフトウェア面での特長と、高いメモリバンド幅というハードウェア面での特長を併せ持つシステムになっています。

SX-Aurora TSUBASA では、Linux OS の環境下で、指定されたプロセスだけが物理的に別の装置である VE で動作します。それぞれのプロセスはもともと他のプロセスから切り離された存在ですので、物理的に別のところで動作していても使い勝手に与える影響はほとんどありません。ユーザ空間自体が VE 側のメモリに確保され、システムコール以外は VE 側でプロセスの実行が完結しますので、VH と VE で大きなサイズのデータを送受信しなければならない場面は限られています。具体的にはノード間通信やファイルアクセスが、大きなデータを送受信する必要のある処理の典型例です。SX-Aurora TSUBASA では、これらの処理での性能への悪影響を最小限に抑えるために、いくつかの技術的な工夫が採られています。例えば、MPI 通信 (ノード間通信) の遅延は並列処理の効率に大きな影響を与えるため、NEC から SX-Aurora TSUBASA 用の MPI 実装が提供されており、システムコールを介さずに VE がネットワーク装置 (Infiniband Host Channel Adapter) と直接データのやり取りをできる (Direct Memory Access, DMA) ようになっています。また、効率の良いファイルアクセスのために、NEC Scalable Technology File System (ScaTeFS) と

いう分散・並列ファイルシステムが提供されています。

第 1 章のポイント

- SX-Aurora TSUBASA はベクトルホスト (VH) とベクトルエンジン (VE) という 2 種類のプロセッサを搭載し、前者が OS 機能、後者が演算を担当しています。
- VE はベクトルアーキテクチャを採用し、高メモリバンド幅とベクトル命令を提供しています。
- VE が得意な計算の特徴として、以下の 3 つが挙げられます。
  1. ベクトル命令を使って実行される。
  2. 3 コア以上を使って実行される。
  3. 再利用性の高いデータがキャッシュに保持される。
- VH 上で動作する OS の機能を使って、VE でプロセスを実行することができます。VH 側の OS 上でプログラムを実行しているような感覚で、VE のハードウェアが提供する性能を享受することができます。

第 2 章

プログラミング環境の概要

本章では、SX-Aurora TSUBASA 向けのプログラミング環境を概説します。SX-Aurora TSUBASA の VE で動作するプログラムは標準的な C/C++ や Fortran を使って記述することができます。新しい/特別なプログラミング言語を覚える必要はありません。また、OpenMP や MPI といった、高性能計算分野で広く用いられている標準的な並列プログラミングモデルも使うことができます。本章では、そのプログラムの記述からコンパイル、実行までの一連の手順を説明します。

本章で登場するコマンド (`vecmd`、`ncc` や `ve_exec`) は標準では `/opt/nec/ve/bin` の下にインストールされていますので、コマンドが見つからない (`command not found`) というエラーメッセージが表示される場合には、環境変数 `PATH` に `/opt/nec/ve/bin` を加えてから再度試してください。Linux 標準の環境 (`bash`) であれば、以下のシェルコマンドを実行します。~`/.bashrc` などの設定ファイルにこの一行を書いておけば、シェルを起動したときに `PATH` が自動設定されますので、毎回手動で設定する必要はありません。

```
$ export PATH=/opt/nec/ve/bin:$PATH
```

なお、本文書のコマンド入力の例で出てくる$はコマンドプロンプトを表しているので、入力する必要はありません。

2.1 システム構成の確認

SX-Aurora TSUBASA は、図 1.4 で示したとおり、VH と VE という 2 種類のプロセッサを混載するシステム構成になっています。一つの VH に対して、複数の VE が接続されているかもしれません。複数の VE がある場合には、それぞれの VE にデバイス ID を割り当てて管理しています。

まずは SX-Aurora TSUBASA のシステム構成を確認しましょう。確認には `vecmd` コマンドを使用します。まずは一般的な Linux コマンドと同様に`-h` をつけて実行して、使用方法を表示させてみましょう。

```
$ vecmd -h
Vector Engine MMM-Command v1.1.3
Usage: vecmd -h [MMM-COMMAND] [command-parameters...]

 {MMM-COMMAND} [command-parameters...]
```

```
--------------+--------------------------------------
 info          [type]
 topo          [control] [option]
```

コマンドラインオプションとして、MMM-COMMAND なるものとそのパラメータを指定できることがわかります。MMM は monitoring and maintenance manager の略です。MMM-COMMAND ごとの使用方法の説明もあります。例えば topo であれば、以下のように使用方法が表示されます。

```
$ vecmd -h topo
Vector Engine MMM-Command v1.1.3
topo: vecmd [-Nh] topo [control] [option]
  A VE card composition information list on the hosts is indicated.

  option
   -N           VE_no is directed.
   -h           help is indicated.

  parameter
   control:
     MATRIX     show the composion of ve-card by matrix-format
     TREE       show the composion of ve-card by tree-format
     ROUTE      show the pci route of ve-cards
     PCI        show the pci infomation of ve-cards

   option:
     -h        distance in MATRIX by hops.
     -r        distance in MATRIX by route.
     -v        pci-info in PCI by more detail.
     -x        pci-info in PCI by config value.

  end status
   Success:
   Failed:      Type miss
```

システム構成を表示させるためには MMM-COMMAND として topo、そのパラメータとして tree、matrix、route、pci のいずれかを指示します。例えば、topo にパラメータとして tree を指定した場合には、システム構成が以下のように表示されます。

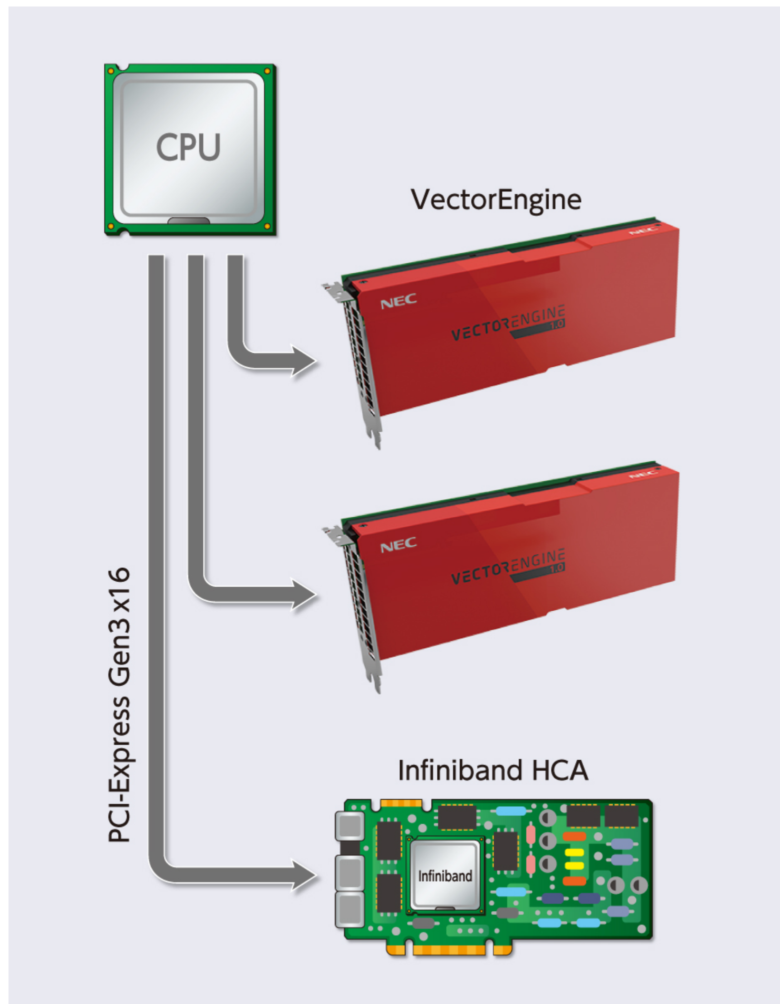


図 2.1　システム構成の例

```
$ vecmd topo tree
Vector Engine MMM-Command v1.1.3
Command:
topo -N 0,1 tree
--------------------------------------------------------------------------------
X11SPG-TF
(Skylake Link)
-+--+-64:00.0---65:00.0 [VE0] [SOCKET0]
    +-16:00.0---17:00.0 [VE1] [SOCKET0]
    +-b2:00.0---b3:00.0 [IB0] [SOCKET0] mlx5_0

--------------------------------------------------------------------------------
Result: Success
```

図 2.1 に示すとおり、このシステムの PCI-Expresss スロットには VE が 2 枚と、InfiniBand の HCA(Host Channel Adapter) が刺さっていることが tree 形式で表示されています。

MMM-COMMAND が `info` の時、`vecmd` は対象となる VE の状態を表示します。VE が複数搭載されている場合には、どの VE の情報を知りたいのか指示することもできます。例えば 2 枚目の VE の情報だけを知りたい場合には、`-N` オプションでデバイス 1 を指示します。

```
$ vecmd -N 1 info
Vector Engine MMM-Command v1.1.3
Command:
info -N 1
--------------------------------------------------------------------------------
Time                            : 2018/06/06 14:44:01
Attached VEs                    : 2
MMM Version                     : 1.1.3
[VE1 : ALL  ]
VE State                        : ONLINE
VE Model                        : 1
Product Type                    : 131
Cores                           : 8
(以下省略)
```

また、`-N` オプションを指定しなかった場合には、すべてのデバイスの情報が表示されます。

SX-Aurora TSUBASA は VH と VE から構成されていますが、以上のように、VE に関する情報も VH 上で動作する Linux 環境から取得できることがわかります。上述の `vecmd` を管理者権限 (root 権限) で実行すると、さらに多くの機能を使うことが可能であり、搭載されている VE の状態を制御することもできます[*1]。この `vecmd` に限らず、本章の以下で説明するように、VE に関するほとんどすべての制御・管理を VH 上の Linux 環境から行うことができます。

2.2 プログラムのコンパイルと実行

SX-Aurora TSUBASA 向けのプログラム開発において、特別なプログラミング言語や統合開発環境を使う必要はありません。例えば古くから用いられている `vi` や `emacs` など、任意のテキストエディタでプログラムを記述し、それをコンパイルして実行ファイルを生成することができます。SX-Aurora TSUBASA 向けのコンパイラとしては、NEC から C/C++ コンパイラと Fortran コンパイラが提供されており、C11/C++14 と Fortran2003[*2] という規格がサポートされています。

まず、以下の C プログラムが `matmul.c` という名前で保存されていると仮定しましょう。このプログラムはただのテキスト形式のファイルなので、どのようなソフトウェアを使って作成しても構いません。

Listing 2.1 単純な行列積プログラム

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #define N (2048)
```

[*1] 管理者権限で `vecmd -h` を実行することで、利用可能な機能が表示されます。

[*2] Fortran2008 の一部の機能もサポートされています。

```
4
5 int main(int argc, char** argv) {
6   int i,j,k;
7   double *a, *b, *c;
8
9   a = (double*)malloc(N*N*sizeof(double));
10  b = (double*)malloc(N*N*sizeof(double));
11  c = (double*)malloc(N*N*sizeof(double));
12
13  /* 行列の初期化 */
14  for(j=0;j<N;j++){
15    for(i=0;i<N;i++){
16      a[j*N+i]=i*j;
17      b[j*N+i]=(i==j?1:0);
18      c[j*N+i]=0.0;
19    }
20  }
21
22  /* 行列積の計算 */
23  for(k=0;k<N;k++){
24    for(j=0;j<N;j++){
25      c[k*N+j] = 0.0;
26      for(i=0;i<N;i++){
27        c[k*N+j] += a[k*N+i]*b[i*N+j];
28      }
29    }
30  }
31
32  return 0;
33 }
```

図 2.2 に示すとおり、このプログラムは $N \times N$ の行列同士の積を計算します。まず、行列用配列のメモリを確保して、最初の二重ループで適当な行列要素を代入しています。その後、三重ループで行列積の計算を行っています。この例では行列 b が単位行列になっているので、最終的に行列 c は行列 a と等しくなっているはずです。C 言語のプログラミング経験がある人であればすぐに理解できる、簡単なプログラムだと言えるでしょう。

図 2.3 に示すとおり、リスト 2.1 のプログラムを Linux 環境で gcc コマンドを使ってコンパイルすれば、VH 側で実行される普通の実行ファイルが生成されます。すなわち、Linux のコマンドライン上で以下の手順で実行されたプログラム matmul は、通常の Linux のプロセスとして VH 上で実行されます。

```
$ gcc matmul.c -o matmul
$ ./matmul
```

これに対して、VE 上でプログラムを実行するためには、VE 用にプログラムをコンパイルする必要があります。VE 用のコンパイラのコマンドは C コンパイラが ncc、C++ コンパイラが nc++、Fortran コンパイラが nfort というコマンドになっています。VE 用のコンパイラが生成した実行ファイルは、自動的に VE 上で実行されます。すなわち、

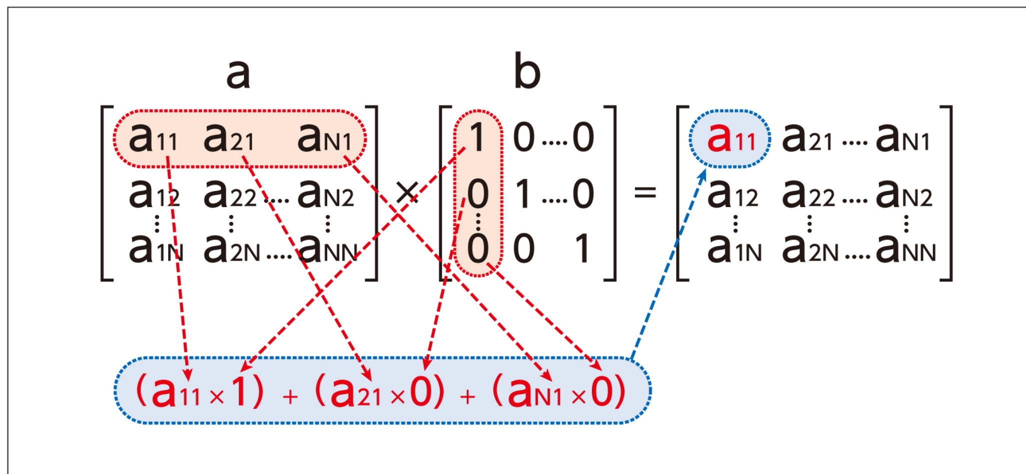


図 2.2　行列積プログラムの動作

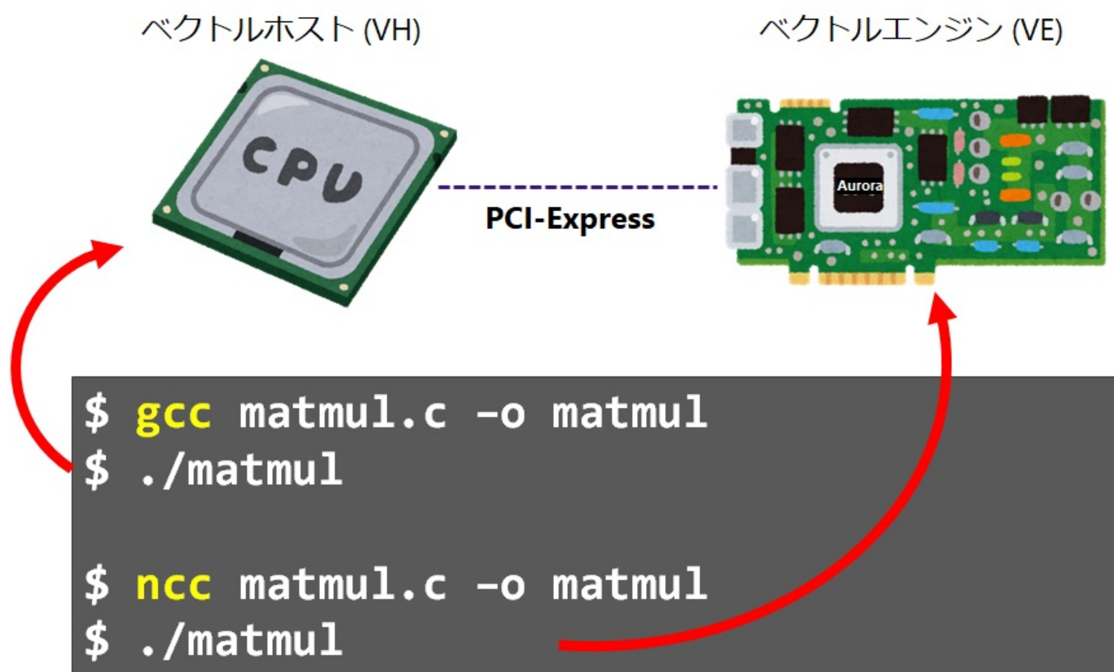


図 2.3　コンパイルと実行

VE 用にプログラムをコンパイルして実行するためには、Linux のコマンドライン上で以下のように入力します*3。

```
$ ncc matmul.c -o matmul
$ ./matmul
```

*3 この例では ncc を実行したときにコンパイラからのメッセージが表示されるかもしれませんが、コンパイルエラー以外のメッセージであれば、とりあえず本章では無視して構いません。それらのメッセージに関しては第 4 章で説明します。

つまり、リスト 2.1 のプログラム例のように、標準的な C/C++ や Fortran で書かれたプログラムを VE 用にコンパイルするだけで、そのプログラムを VE 上で実行することができます。単に使うだけなら標準的なLinux 環境でプログラムを通常通りコンパイルして実行する場合とほとんど変わらない労力で、VE を利用できると言えます。

複数の VE が搭載されているシステムでは、ve_exec コマンドを利用することでどの VE で実行するのかを指示することもできます。例えばデバイス 1 で実行したい場合には以下のように指示します。

```
$ ve_exec -N 1 ./matmul
```

また、デバイス ID が指定されていないときにはデバイス 0 が使われますが、環境変数 VE_NODE_NUMBER を設定することで、デフォルトのデバイスを変更することができます。例えば以下のコマンドを実行すると、./matmul がデバイス 1 で実行されます。

```
$ export VE_NODE_NUMBER=1
$ ./matmul
```

以下、ncc や nc++ で利用可能な主なコンパイラオプションを紹介します。ほとんどのオプションが GNU のコンパイラ (gcc や g++) と変わらないことがわかります。他にもコンパイラオプションはありますが、本文書では必要になったところで紹介します。

主な全体オプション

-c
リンクは行わずにオブジェクトコード (.o) を出力する。

-fsyntax-only
文法チェックのみを行う。

-o *filename*
出力ファイル名を指定する。

-x *language*
入力ファイルで使われているプログラミング言語を明示的に指定する。
language には c や c++ を指定できる。

主な最適化オプション

-O[*n*]
最適化レベルを指定する (デフォルトは-O2)。

主なコード生成オプション

-fpic | -fPIC
Position-independent code(共有オブジェクト用のコード) を生成する。

`-ftrace`
Ftrace 機能 (第 3 章参照) を利用可能なコードを生成する。

主な警告オプション

`-Wall`
すべての警告メッセージを出力する。

主なデバッグオプション

`-g`*level*
デバッグ情報を生成するようにコンパイルする (*level* は省略可能で、デフォルトは`-g2`)。第 2.3 節の `gdb` を使うためには、このオプションを指定する必要がある。

主なリスト出力オプション

`-report-diagnostics`
診断メッセージリスト (第 4 章参照) を出力する。
`-report-format`
編集リスト (第 4 章参照) を出力する。
`-report-all`
診断メッセージリストと編集リスト (第 4 章参照) の両方を出力する。

主なプリプロセッサオプション

`-E`
プリプロセッサだけを適用して結果を出力する。
`-I`*directory*
ヘッダファイルのディレクトリを指定する。

主なリンカオプション

`-L`*directory*
ライブラリファイルのディレクトリを指定する。
`-l`*library*
リンクするライブラリを指定する。
例えば `libfoo.so` をリンクする場合には`-lfoo` となる。
`-shared`
共有オブジェクト (`.so`) を生成する。

なお、実行ファイルが VE 用にコンパイルされているかどうかを確認するためには、`nreadelf` コマンドを使います。

```
$ /opt/nec/ve/bin/nreadelf -h ./matmul
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
```

```
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           NEC VE architecture
  Version:                           0x1
  Entry point address:               0x600000001700
  Start of program headers:          64 (bytes into file)
  Start of section headers:          80608 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         7
  Size of section headers:           64 (bytes)
  Number of section headers:         29
  Section header string table index: 26
```

VE 用にコンパイルされている場合には、Machine の行に NEC VE architecture と表示されます。VE に対応している以外、`nreadelf` は標準の Linux コマンドである `readelf` と同じですので、詳しい使い方は Linux の man ページ等を参照してください。

2.3 プログラムのデバッグ

プログラムを作成していると、実行したときの動作が期待とは異なる場合がよくあります。そのような場合には、期待通りの動作をするようにプログラムを書き換える必要があり、その作業を一般的にデバッグ (debug) といいます。SX-Aurora TSUBASA では、Linux 環境で標準的に用いられているデバッグ用ツール (**デバッガ** (debugger)) を使って、VE 向けのプログラムのデバッグ作業をすることができます。具体的には、`/opt/nec/ve/bin` の下にある `gdb` コマンドを使って、従来通りの `gdb` の使い方で VE 向けプログラムのデバッグができます。この第 2.3 節には SX-Aurora TSUBASA 特有の内容はほとんどありませんので、すでに `gdb` の使い方を知っている人は第 2.3 節を読み飛ばしても構いません。

デバッガの主な機能として、以下の項目が挙げられます。

ブレークポイント (breakpoint) デバッガは、プログラムの実行を任意の段階で中断することができます。中断するプログラム中の位置のことをブレークポイントと呼びます。

ステップ実行 (step execution) ブレークポイントで中断した時点から、プログラムを一行ずつ実行することができます。そのように少しずつ段階的に実行していく機能をステップ実行と呼びます。

ワッチポイント (watchpoint) ある変数の値が変化したときに、プログラムの実行を中断する機能もあります。そのように状態を監視する対象のことをワッチポイントと呼びます。VE 向けの `gdb` コマンドではこのワッチポイント

の機能は現在サポートされていません。

変数の表示と変更　プログラムの実行を中断しているとき、各種変数の値を表示したり、変更したりすることができます。それによって、中断した時点でのプログラムの状態を確認することができます。

一般的なデバッグ作業では、これらの機能を使いながら実行中のプログラムの各種状態を確認し、プログラムの問題点 (バグ) を発見して、正しい挙動となるようにその問題点を修正していきます。

プログラム (以下の例では matmul.c) をデバッグするためには、まず-g オプションをつけてそのプログラムをコンパイルします。

```
$ ncc -g matmul.c -o matmul
```

次に、コンパイルの結果生成された実行ファイルを引数として gdb コマンドを実行することで、そのプログラムをデバッグすることができます。gdb を起動するとメッセージが表示され、(gdb) というプロンプトが表示されて入力待ち状態になります。

```
$ /opt/nec/ve/bin/gdb matmul
GNU gdb (GDB) 7.11.1-12.el7
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=x86_64-pc-linux-gnu --target=ve-nec-linux".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from matmul...done.
(gdb)
```

ここでコマンドを入力することでデバッグ作業を行います。この環境を以下の説明では **GDB 環境**と呼びます。GDB 環境で利用可能な主なコマンドは以下のとおりです。

r

対象となるプログラムを実行します。このコマンドの引数が、コマンドライン引数としてプログラムに渡されます。例えば、gdb ./matmul で起動した GDB 環境で r 10 と入力すると、通常のコマンドラインで./matmul 10 と実行

した場合と同じ動作をします。

`b`

対象となるプログラム中にブレークポイントを設定します。プログラムを実行すると、ブレークポイントの位置で実行が中断しますので、その位置でのプログラムの状態を調査することができます。`b` *function* という形式で関数名を指定すると、その関数が呼び出されるたびにブレークします。`b` *filename:line* という形式で、ソースコード中の特定の行にブレークポイントを設定することもできます。

`i`

各種情報を表示します。例えば、`i b` で現在設定されているブレークポイントの一覧を表示します。

`d`

`d` *num* で、*num* に対応する現在設定されているブレークポイントを削除します。

`n`

ブレークポイントでプログラムの実行が中断している状態で入力すると、プログラムを一行分だけステップ実行します。関数呼び出しがある場合、その関数内には移動しないでステップ実行を続けます。

`s`

ブレークポイントでプログラムの実行が中断している状態で入力すると、プログラムを一行分だけステップ実行します。関数呼び出しがある場合、その関数内に移動してステップ実行を続けます。

`c`

次のブレークポイントまでプログラムの実行を進めます。

`p`

ブレークポイントで中断しているときに `p` *expression* という形式で使用すると、プログラム中の *expression* の現在の値を表示します。

`bt`

ブレークポイントで中断しているときに使用すると、どのような関数呼出しで現在のブレークポイントに到達したのか、その経緯を出力します。さらに `up` コマンドと `down` コマンドを使って、その関数呼出しのつながりの中を移動することができます。`up` コマンドは、呼び出し元の関数の呼び出し位置に移動し、`down` コマンドは逆に呼び出し先の関数に移動します。

`l`

対応する位置のソースコードを表示します。`l` *linenum* と指示することで、*linenum* 周辺のソースコードを表示することもできます。

`q`

GDB 環境を終了します。

他にも多くのコマンドが用意されていますが、上記だけでもかなり詳細にプログラムの挙動を調べることができます。以下に、gdb コマンドの使用例を示します[*4]。

```
$ /opt/nec/ve/bin/gdb matmul
GNU gdb (GDB) 7.11.1-12.el7
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=x86_64-pc-linux-gnu --target=ve-nec-linux".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from matmul...done.
(gdb) b foo
Breakpoint 1 at 0x600000001918: file matmul.c, line 6.
(gdb) b 28
Breakpoint 2 at 0x600000001d90: file matmul.c, line 28.
(gdb) r
Starting program: /opt/nec/ve/bin/ve_exec --traceme  -- /home/veuser/matmul

Breakpoint 1, foo () at matmul.c:6
6       }
(gdb) c
Continuing.

Breakpoint 2, main (argc=1, argv=0x60ffffff538) at matmul.c:28
28              for(k =0;k<N;k ++){
(gdb) l
23                              c[j*N+i]=0.0;
24                      }
25              }
26
```

*4 この例では GDB 環境の使い方を示すために、matmul.c の 4〜6 行目に空の関数 foo を追加し、それを関数 main から呼び出すように改変したプログラムを使用して説明しています。

```
27              /* 行列積の計算*/
28              for(k =0;k<N;k ++){
29                      for(j=0;j<N;j++){
30                              c[k*N+j] = 0.0;
31                              for(i=0;i<N;i++){
32                                      c[k*N+j] += a[k*N+i]*b[i*N+j];
(gdb) p a
$1 = (double *) 0x60000c000010
(gdb) q
A debugging session is active.

        Inferior 1 [process 24492] will be killed.

Quit anyway? (y or n) y
```

上記の例では、最初に `b` コマンドで 6 行目にある関数 `foo` と、28 行目にブレークポイントを設定しています。その後に、`r` コマンドでプログラムを実行しています。ブレークポイント (この例では関数 `foo`) で実行が中断します。`c` コマンドで実行を再開すると、次のブレークポイント (28 行目) で実行が中断するので、`l` コマンドで中断した位置の周辺のソースコードを表示しています。また、`p` コマンドで実行が中断した時点での変数 `a` の値を表示しています。最後に、`q` コマンドで GDB 環境を終了しています。

実行中のプロセスをデバッグする場合には、実行ファイルを指定して gdb コマンドを起動し、GDB 環境で attach コマンドでプロセス ID を指定します。通常のプロセスと同様に VE 上で動作するプロセスにもプロセス ID が割り当てられ、`ps` コマンド等で確認することができます。

第 2 章のポイント

- VH 上の Linux 環境から、SX-Aurora TSUBASA のシステム構成を確認したり、搭載されている VE を管理・制御したりすることができます。
- コンパイラとして `ncc`、`nc++`、`nfort` を使い、生成された実行ファイルを `ve_exec` コマンド経由で起動すると、VE でプログラムが実行されます。
- VE 上で動作するプログラムのデバッグには標準的な `gdb` コマンドを利用できます。

第 3 章

性能解析

高性能計算プログラミングにおいて、性能解析は非常に重要です。一般的に、実行時間の大部分はプログラムのほんの一部を実行するために費やされています。その時間がかかる部分を高速化できれば、プログラム全体の実行時間を大幅に短縮できます。プログラムのどこの部分が、どのような理由で実行時間を費やしているのかを知ることは、効果的な最適化の方針を考える上で必要不可欠です。第 3 章では、SX-Aurora TSUBASA 向けの高性能計算プログラミングを考える上で重要なプログラムの性能指標を紹介し、その性能解析のために提供されている 2 つのツールの使い方を説明します。1 つは PROGINF、もう 2 つは FTRACE と呼ばれています。

3.1 性能指標

SX-Aurora TSUBASA でプログラムを実行したときの性能の特性を定量的に表すため、いくつかの指標が使われます。以下では、後述の性能解析ツールで取得できる性能指標を説明します。

OPS 値　1 秒あたり実行された演算数を表す指標です。SX-Aurora TSUBASA の性能解析ツールで表示される MOPS は Million Operations Per Second の略であり、例えば 1 MOPS であれば毎秒 1,000,000 回の演算を行ったことを表しています。

FLOPS 値　1 秒あたり実行された浮動小数点演算数を表す指標です。SX-Aurora TSUBASA の性能解析ツールで表示される MFLOPS は Million FLoating-point Operations Per Second の略であり、例えば 1 MFLOPS であれば毎秒 1,000,000 回の浮動小数点演算を行ったことを表しています。

ベクトル演算率　プログラム中のすべての演算の中で、ベクトル演算命令で実行された演算の割合を**ベクトル演算率** (Vector Operation Ratio, VOR) と呼びます。これは式 (3.1) で表されます。

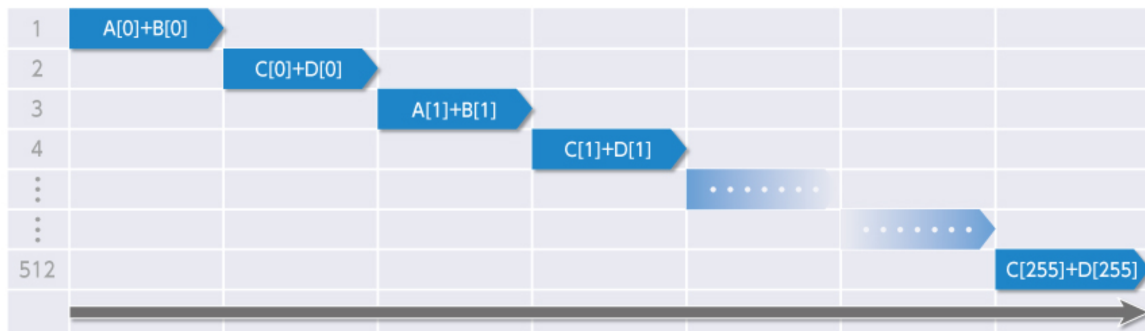
$$\alpha = N_{\mathrm{vec}}/N_{\mathrm{total}} \tag{3.1}$$

ここで、N_{vec} はベクトル演算命令で実行される演算数、N_{total} は総演算数です。図 3.1 に示すとおり、ベクトル演算命令で実行可能な演算をベクトル演算命令で実行したほうが実行時間を短くできると期待されるため、ベクトル演算率 α は高いほうが性能向上を期待できます。

参考までに、より直接的に実行時間を使った指標として、**ベクトル化率** (Vectorization Ratio, VR) もあります。これは式 (3.2) で定義されます。

$$\alpha' = T_{\mathrm{vec}}/T_{\mathrm{total}} \tag{3.2}$$

ここで、T_{vec} はベクトル化可能な演算をすべてスカラ演算命令で実行した場合の時間、T_{total} はすべての演算をスカラ演算命令で実行した場合のプログラム全体の総実行時間です。$\alpha \simeq \alpha'$ であり、ベクトル演算率がベクトル化率と同義に使われる場合もあるのですが、両者は正確には一致しません。現代の複雑化したプロセッサで



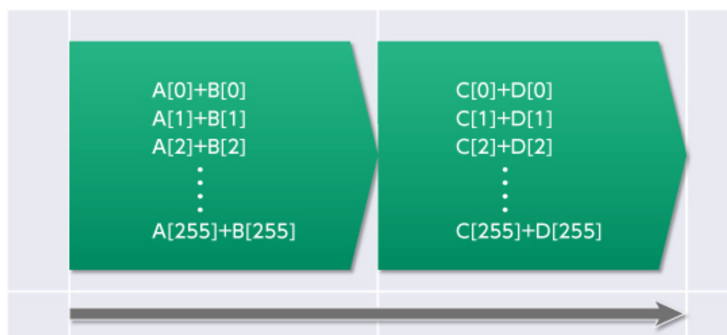


図 3.1　スカラ演算命令とベクトル演算命令の実行イメージ

は式 (3.2) のベクトル化率を正確に計測するのは難しいので、式 (3.1) のベクトル演算率の方が主に使われています。

平均ベクトル長　ベクトル命令を実行するとき、その実行開始までにはある一定の時間が必要です。これをベクトル命令の**立上がり時間** (startup time) と呼びます。ベクトル命令の実行がいったん始まってしまえば、ベクトルアーキテクチャは効率よくベクトル処理を行うことができます。しかし、ベクトル処理がすぐに終わってしまうような場合には、立上がり時間の分だけ実行時間が却って長くなってしまうかもしれません (図 3.2 参照)。したがって、ベクトル命令を効果的に使うためには、ベクトルがある程度の長さ (**交差ループ長**) 以上であることが重要です。

プログラム中でベクトル命令が処理した演算要素数 (ベクトルの長さ) の平均を、**平均ベクトル長**と呼びます。SX-Aurora TSUBASA の場合、1 つのベクトル命令で処理可能な最大ベクトル長は 256 なので、平均ベクトル長が 256 に近い場合にはベクトル命令が効率よく実行されていると考えることができます。一方、それよりも極端に短い場合には、立ち上がり時間の影響でベクトル命令の実行が非効率になっている可能性があります。そのような場合、平均ベクトル長を大きくするようにプログラムを最適化することで、そのプログラムの性能を向上させる余地があることがわかります。

キャッシュミス時間　VE にはキャッシュが搭載されており、メモリ上のデータよりもキャッシュ上にあるデータの方が短い**アクセス遅延時間 (レイテンシ)** でアクセスできます。アクセスしたいデータがキャッシュ上に無い状況のことを**キャッシュミス**と呼び、逆にキャッシュ上に必要なデータがあることを**キャッシュヒット**と呼びます。キャッシュミス時にはメインメモリ上のデータを読み書きすることになりますので、キャッシュヒット時と比較して長い遅延時間が必要になります。キャッシュミスによって、キャッシュヒット時と比較して余分に必要に

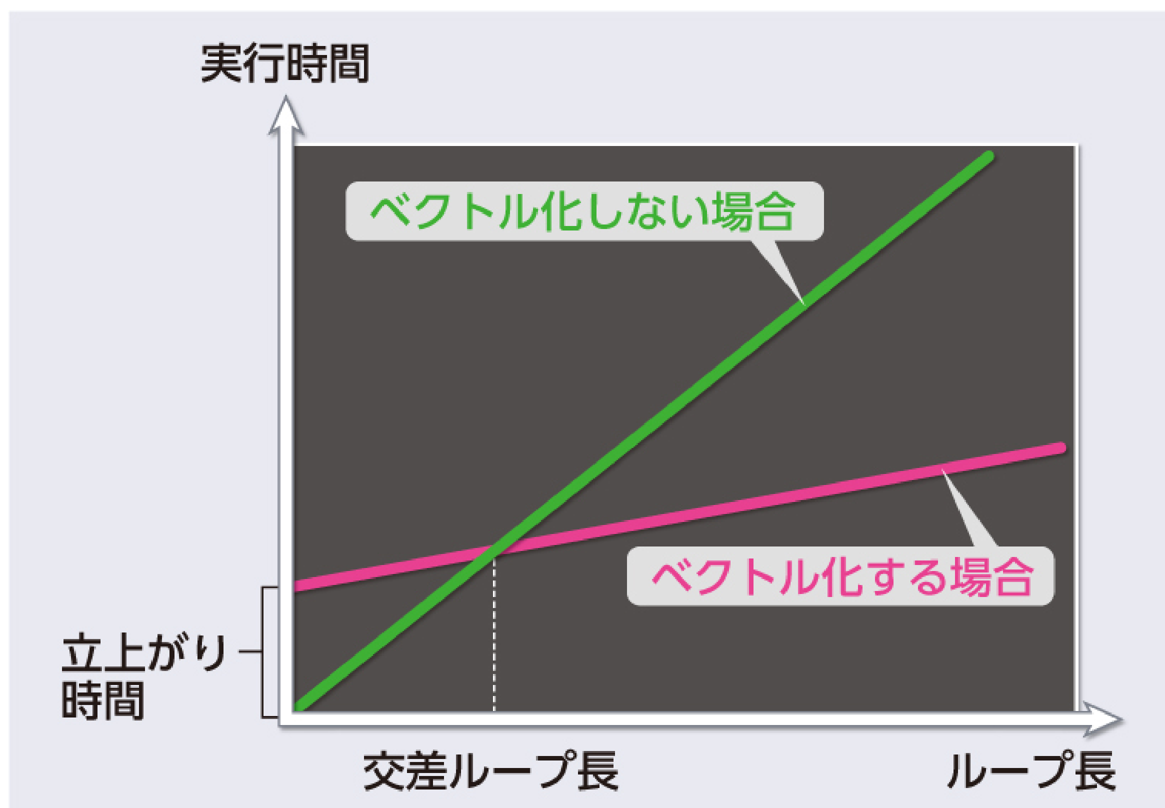


図 3.2　立上がり時間と交差ループ長

なった時間を表しています。キャッシュミスをして長いアクセス遅延時間が生じたとしても、その間にベクトル演算器が他の演算を実行していて結果的にプログラムの総実行時間が増加しない場合には、キャッシュミス時間は増えません。VE には**レベル *1* キャッシュ** (L1 キャッシュ) と**ラストレベルキャッシュ** (Last-Level Cache, *LLC*) があり、後述のプロファイル情報では L1 キャッシュにおけるキャッシュミス時間が表示されます。

3.2　プログラム全体の性能情報

高性能計算プログラムの性能解析において、一番最初に行うべきことはプログラム全体の性能情報を調べることです。これは容易に調べることができますし、その結果として得られる統計情報を見ることで、性能改善の余地がどの程度あるのかを推測することもできます。SX-Aurora TSUBASA では、そのようなプログラム全体の性能に関する統計情報を調べるための手段として、PROGINF というツールが提供されています。

実行時に環境変数 `VE_PROGINF` を `YES` や `DETAIL` に設定することで、PROGINF 情報が表示されます。例えば、リスト 2.1 に示した `matmul.c` をコンパイルしてその性能の統計情報を得る場合には、以下のようにコンパイルして実行します。

```
$ ncc matmul.c -o matmul
$ export VE_PROGINF=DETAIL
$ ./matmul
```

これによって、プログラムの実行終了時に以下のような性能統計情報が表示されます。

```
          ********  Program  Information  ********
  Real Time (sec)                         :             58.430857
  User Time (sec)                         :             58.418652
  Vector Time (sec)                       :             58.411956
  Inst. Count                             :             520823934
  V. Inst. Count                          :             109238296
  V. Element Count                        :           27964998424
  V. Load Element Count                   :           17179869443
  FLOP Count                              :           18253611020
  MOPS                                    :            632.855700
  MOPS (Real)                             :            632.663528
  MFLOPS                                  :            312.496340
  MFLOPS (Real)                           :            312.401448
  A. V. Length                            :            255.999951
  V. Op. Ratio (%)                        :             98.886599
  L1 Cache Miss (sec)                     :              0.000162
  CPU Port Conf. (sec)                    :              5.752188
  V. Arith. Exec. (sec)                   :              0.336024
  V. Load Exec. (sec)                     :             57.944100
  VLD LLC Hit Element Ratio (%)           :             49.601852
  Power Throttling (sec)                  :              0.000000
  Thermal Throttling (sec)                :              0.000000
  Memory Size Used (MB)                   :            348.000000

  Start Time (date)        :        Sun Apr  7 17:36:03 2019 JST
  End   Time (date)        :        Sun Apr  7 17:37:02 2019 JST
```

PROGINF によって表示される各項目の値を見ることで、性能が低い原因の予測や性能改善の余地の有無の確認をすることができます。この例では、後述の `V. Load Exec.` でほとんどの時間を費やしており、メモリアクセスの効率が悪いということを読み取ることができます。このため、メモリアクセスの効率が良くなるようにプログラムを最適化していくことになります[*1]。

`Real Time (sec)` : 実時間

プログラム全体の実行時間 (実際の経過時間) を秒単位で表示します。

`User Time (sec)` : ユーザ時間

[*1] この例では、コンパイラオプションとして `-O3` を追加し、コンパイラ最適化を有効にしてコンパイルする (`ncc -O3 matmul.c -o matmul`) ことで性能が劇的に改善します。

プログラム全体の実行時間のうち、ユーザプロセスの実行に費やした時間を秒単位で表示します。この時間と実時間 (Real Time) との差が、カーネルの実行 (システムコール) 等に費やした時間になります。差が大きい場合には、必要以上にシステムコールをしていないか調査する、という方針を立てることができます。

Vector Time (sec) : ベクトル時間

プログラム全体の実行時間のうち、ベクトル命令の実行に費やした時間を秒単位で表示します。この時間が短い場合には、後述のベクトル演算率 (V. Op. Ratio) が低いことが予想されます。一般に、ベクトル演算率が低い場合にはより多くのループをベクトル化することを考えます。

Inst. Count : 命令数

実行された命令数の合計を表しています。

V. Inst. Count : ベクトル命令数

実行されたベクトル命令数の合計を表しています。

V. Element Count : ベクトル要素数

ベクトル演算命令で演算されたベクトル要素数の合計を表しています。

V. Load Element Count : ベクトルロード要素数

ベクトルロード命令やベクトルギャザー命令で読み込まれたベクトル要素数の合計を表しています。

FLOP Count :浮動小数点演算数

実行された浮動小数点演算の数の合計を表しています。

MOPS と MOPS (Real) これら 2 つの指標は両方とも MOPS 値を表しています。前者はユーザ時間、後者は実時間を分母として、1 秒あたりの演算数が計算されています。

MFLOPS と MFLOPS (Real) これら 2 つの指標は両方とも MFLOPS 値を表しています。前者はユーザ時間、後者は実時間を分母として、1 秒あたりの浮動小数点演算数が計算されています。

A. V. Length : 平均ベクトル長

ベクトル命令の処理対象となったベクトルの平均の長さを表しています。VE のベクトル演算命令は最長で 256 要素のベクトルを処理します。ベクトル化されるループが短い場合はもちろんのこと、256 より長いループを実行する場合でも平均ベクトル長は小さくなりえます。例えば、長さ 300 のループをベクトル化した場合、256 の長さのループと 44 の長さのループに分けて (ストリップマイニング) から実行され、平均ベクトル長は 150 になります。平均ベクトル長が 256 よりも大幅に小さい場合には、プログラムに対してベクトル長を伸ばすような最適化を施すことによって性能が改善する可能性があります。

V. Op. Ratio (%) : ベクトル演算率

ベクトル演算率を表しています。VE の場合、スカラ演算性能よりもベクトル演算性能のほうが遥かに高いため、できるだけベクトル演算命令を使ってプログラムを実行したほうが高い実効性能を達成できます。このため、一般的にベクトル演算率は高ければ高いほど良いといえます。VE の性能を引き出すためには、ベクトル演算率は 100% 近い値 (できれば 99% 以上) にする必要があります。

L1 Cache Miss (sec) : L1 キャッシュミス時間

L1 キャッシュに必要なデータが無かった場合、キャッシュミスとなります。キャッシュミス時には必要なデータがメモリからキャッシュへコピー (フェッチ) されます。このキャッシュミスによって余分に必要となった時間の合計を秒単位で表しています。キャッシュミスをしても、フェッチしている間に別の演算を実行することで結果的に実行時間が伸びなかった場合には、キャッシュミス時間にはカウントされません。

CPU Port Conf. (sec) : CPU ポート衝突時間 (DETAIL 時のみ表示)

第 5 章で後述するとおり、プロセッサ (CPU) の各コアにはデータの入出口 (ポート) があります。各コアあたり 16 個のポートが用意されており、それぞれ LLC の一部と対応付けられています。同時に複数のメモリアクセス要求が同じポートを使おうとする場合、1 つのメモリアクセス要求だけがポートを利用することができ、他のメ

モリアクセス要求はポートが空くまで待たされます。このように、同時に同じ CPU ポートを使おうとしてメモリアクセスが遅延する現象を *CPU* **ポート衝突** (conflict) と呼びます。CPU ポート衝突時間が顕著に長い場合には、メモリアクセスパターンの変更を検討することで高速化を目指します。

`V. Arith. Exec. (sec)` : ベクトル演算実行時間 (`DETAIL` 時のみ表示)
: ベクトル演算の実行に費やした時間を秒単位で示します。

`V. Load Exec. (sec)` : ベクトルロード実行時間 (`DETAIL` 時のみ表示)
: ベクトルロードの実行に費やした時間を秒単位で示します。

`VLD LLC Hit Element Ratio (%)` :LLC ヒット要素率
: ベクトルロードの際に、必要なデータが LLC に保持されていた割合を示します。この値が高ければ、LLC に再利用可能なデータをうまく保持できていることがわかります。

`Power Throttling (sec)` :電力スロットリング時間 (`DETAIL` 時のみ表示)
: 省電力のためにハードウェアが停止していた時間を秒単位で示します。

`Thermal Throttling (sec)` :熱スロットリング時間 (`DETAIL` 時のみ表示)
: 発熱を抑えるためにハードウェアが停止していた時間を秒単位で示します。

`Max Active Threads` : 最大アクティブスレッド数 (`DETAIL` 時のみ表示)
: スレッド並列化されたプログラムを実行した時に、起動されたスレッド数の最大値を示します。

`Available CPU Cores` :利用可能 CPU コア数 (`DETAIL` 時のみ表示)
: スレッド並列化されたプログラムを実行した時に、利用可能な CPU コア数を示します。

`Average CPU Cores Used` :平均使用コア数 (`DETAIL` 時のみ表示)
: スレッド並列化されたプログラムを実行した時に、実際に使われたコア数の時間平均値を示します。この値が利用可能な CPU コア数 (`Available CPU Cores`) よりも小さい場合には、並列化可能な部分が増えるようにプログラムを最適化することを考えます。

`Memory Size Used (MB)` : 使用メモリサイズ
: プログラムが使用したメモリ容量を表しています。VE には物理的に最大で 48 GB の容量のメモリが搭載されています。この容量を超えるプログラムをそのままでは実行できませんので、使用メモリサイズに配慮しながらプログラムのデータサイズを決める必要があります。

`Start Time (date)` と `End Time (date)` プログラムの実行開始時間と終了時間を表しています。

3.3 詳細な性能情報

PROGINF がプログラム全体の性能に関する統計情報を表示するのに対して、FTRACE はプログラム中の特定部分の性能情報を取得するために用いられます。FTRACE では、関数あるいはプログラマが指示した範囲の性能情報を取得します。その結果として、プログラムのどの部分で実行時間を費やしているのかが明確になり、最適化をすべきプログラムの範囲を知ることができます。ただし、FTRACEを使う場合には、使わない場合と比較して性能が低下しますので、あくまでも性能解析の時にだけ FTRACE を使うようにしましょう。

FTRACE を使うためには、コンパイル時に`-ftrace` オプションを指定する必要があります。このオプションは C コンパイラ (`ncc`)、C++ コンパイラ (`nc++`)、および Fortran コンパイラ (`nfort`) で同様に使えます[*2]。。

[*2] 大規模システムでの並列計算プログラムを開発するための MPI プログラミング [5] でも、FTRACE は利用可能です。その場合には、MPI 用 C コンパイラ (`mpicc`)、MPI 用 C++ コンパイラ (`mpic++`)、および MPI 用 Fortran コンパイラ (`mpif90`) に`-ftrace` オプションを指示する必要があります。ここでは MPI を使わないプログラムの FTRACE の使い方を説明します。

```
$ ncc -ftrace functions.c -o functions
```

コンパイル時に-ftrace オプションをつけた場合、プログラム実行時に ftrace.out というファイルが自動生成されるようになります。このファイルの中に、性能情報が保存されています。その情報を見るためには ftrace コマンドを使います。

文法

ftrace [-all] [-num *n*] -f *filename(s)*

オプション

-all　保存されているすべての関数の性能解析情報を表示します。

-num　実行時間の長い関数を *n* 番目まで表示します。

-f　性能解析情報が保存されているファイル名を指定します。空白で区切って複数のファイル名を指定することもできます。他のオプションで指示されていなければ、実行時間の長い順に 10 位までの関数の性能解析情報を表示します。

この ftrace コマンドの使用例を以下に示します。

```
$ ftrace -f ftrace.out
*----------------------*
  FTRACE ANALYSIS LIST
*----------------------*
Execution Date : Wed Oct 11 16:18:01 2017 JST
Total CPU Time : 00:11:25.234
FREQUENCY  EXCLUSIVE       AVER.TIME     MOPS   MFLOPS  V.OP  AVER.    VECTOR L1CACHE CPU PORT  PROC.NAME
           TIME[sec](  % )    [msec]                    RATIO V.LEN      TIME    MISS     CONF

        8   449.325( 65.6) 56165.608   2306.3     76.3   0.14   5.0     1.003   5.387    0.000  funcA
     1012   134.555( 19.6)   132.960   2209.0    322.6  43.34  17.4    55.742  25.096    0.000  funcB
    62248    40.143(  5.9)     0.645   2210.0   1040.1   0.00   0.0     0.000   0.647    0.000  funcC
    62248    37.709(  5.5)     0.606   2200.2   1026.4   0.00   0.0     0.000   0.532    0.000  funcD
   248992    16.235(  2.4)     0.065    216.6      0.0  52.45   4.7    15.055   0.624    0.000  funcE
     2032     4.834(  0.7)     2.379    415.8      0.0  28.61   6.3     4.098   0.246    0.000  funcF
  3147416     0.647(  0.1)     0.000   4232.6   1036.3  47.46   5.0     0.602   0.008    0.000  funcG
        8     0.606(  0.1)    75.725   1739.6     76.4   6.25   5.0     0.061   0.109    0.000  funcH
        4     0.507(  0.1)   126.798   1190.6      0.0  15.07  24.6     0.111   0.001    0.000  funcI
        4     0.376(  0.1)    94.005   3245.3    600.0  54.80  12.1     0.178   0.058    0.000  funcJ
------------------------------------------------------------------------------------------------------------
  3524029   685.234(100.0)     0.194   2213.2    232.2   6.54  15.7    76.912  32.748    0.000  total
```

実行日時 (Execution Date) や総 CPU 時間 (Total CPU Time) の後に、各種性能情報が関数ごとに一行ずつ表示されています。関数ごとの表示の後に、プログラム全体の値が表示されます。各項目の値の意味は以下の通りです。

PROC.NAME　関数名です。

FREQUENCY　対応する関数が呼び出された回数を示しています。上記の例では関数 funcA がプログラム実行中に 8 回実行されたことを示しています。

EXCLUSIVE TIME　対応する関数の実行に要した時間を秒単位で表しています。() 内は全体の実行時間に対する割合

を % 表記しています。関数 1 の中で別の関数 2 が呼ばれている場合、関数 2 の実行時間は関数 1 の実行時間の中には含まれません。

AVER.TIME　対応する関数の実行に要する平均時間をミリ秒単位で表しています。上述の EXCLUSIVE TIME を FREQUENCY で割った値をミリ秒単位で表しています。

MOPS　対応する関数内での MOPS 値を表しています。

MFLOPS　対応する関数内での MFLOPS 値を表しています。

V.OP RATIO　対応する関数内でのベクトル演算率を表しています。

AVER.V.LEN　対応する関数内での平均ベクトル長を表しています。

VECTOR TIME　対応する関数内でのベクトル命令実行時間を秒単位で表しています。

L1CACHE MISS　対応する関数内でのキャッシュミス時間を秒単位で表しています。

CPU PORT CONF　対応する関数内での CPU ポート衝突時間を秒単位で表しています。

FTRACE が表示する性能情報から、どの関数でどれくらいの実行時間がかかっているかを知ることができます。また、実行時間の長い関数に関して、なぜ実行時間が長いのかを推測することもできます。例えば上述の例では、関数 funcA の実行時間が他の関数と比べて非常に長く、総実行時間の 65% を関数 funcA の実行に費やしていることがわかります。このため、プログラム最適化作業においては関数 funcA の高速化を最優先で考えることになります。性能情報によると、ベクトル演算率や平均ベクトル長、ベクトル時間の値がすべて小さいことがわかります。したがって、この関数の実行にはベクトル命令がほとんど使われておらず、そのために実行時間が長くなっていると推測されます。このため、もしもなるべくベクトル命令が使われるように関数 funcA を修正することができれば、性能の大幅な改善が期待できます。また、例えば関数 funcE や funcG のように、1 回の実行時間が短い関数が、膨大な回数呼ばれることで総実行時間のある程度の割合を費やすことがあります。そのような場合には、関数を**インライン展開**することによって関数呼び出しのオーバヘッドをなくし、総実行時間を短縮するという方針も考えられます。実用的なプログラムは数千行から数十万行におよぶこともあり、ソースコードを読んでいただけでは最適化すべきところを見つけるのは容易ではありませんが、FTRACE のような性能解析ツールを使えば最適化すべき場所を効率よく限定していくことができます。

多くの場合、実行時間のかかる部分はソースコードの中でループとして書かれています。このため、関数が複数のループを含んでいる場合、どのループの実行に時間がかかっているのかを調べたい場合があります。そのように関数の一部の実行時間を調べるために、多くのデバッガでは調べたい範囲を指定する手段が提供されています。FTRACE の場合には、関数 ftrace_region_begin と ftrace_region_end を使います。これらの関数は ftrace.h で以下のように定義されています。

```
/* ユーザ指定範囲の開始位置 */
extern int ftrace_region_begin(const char* id);

/* ユーザ指定範囲の終了位置 */
extern int ftrace_region_end(const char* id);
```

ここで、引数 id は任意の文字列です。範囲の開始位置で使った文字列を終了位置でも使うことで対応関係を表現します。すなわち、同じ文字列を引数として使って関数 ftrace_region_begin と ftrace_region_end を呼び出すことにより、それらの 2 つの関数呼び出しに挟まれた範囲の性能情報が収集されます。

ソースコードの例をリスト 3.1 に示します。

Listing 3.1　ユーザ指定範囲の例

```
1 #include <ftrace.h>
2 ...
3 (void) ftrace_region_begin("loop#1"); // 外側の範囲開始
```

```
 4 for (i = 0; i < n; i++) {
 5   ...
 6 }
 7 (void) ftrace_region_begin("loop#2"); // 内側の範囲開始
 8 for (j = 0; j < n; j++) {
 9   ...
10 }
11 (void) ftrace_region_end("loop#2"); // 内側の範囲終了
12 (void) ftrace_region_end("loop#1"); // 外側の範囲終了
```

関数 ftrace_region_begin と ftrace_region_end を使ってユーザ指定範囲の性能情報を得るためには、コード中に#include <ftrace.h>が必要です。リスト 3.1 のように複数の文字列を使い分けることによって、ネストして (入れ子状にして) 範囲を指定することもできます。

ユーザ指定範囲の性能方法は、FTRACE の表示する性能情報に以下のように現れます。

```
FREQUENCY  EXCLUSIVE       AVER.TIME     MOPS   MFLOPS  V.OP  AVER.    VECTOR L1CACHE CPU PORT  PROC.NAME
           TIME[sec]( % )    [msec]                     RATIO V.LEN      TIME    MISS     CONF

        8   449.325( 65.6) 56165.608   2306.3     76.3   0.14   5.0     1.003   5.387    0.000  funcA
     1012   134.555( 19.6)   132.960   2209.0    322.6  43.34  17.4    55.742  25.096    0.000  funcB
    62248    40.143(  5.9)     0.645   2210.0   1040.1   0.00   0.0     0.000   0.647    0.000  funcC
    62248    37.709(  5.5)     0.606   2200.2   1026.4   0.00   0.0     0.000   0.532    0.000  funcD
   248992    16.235(  2.4)     0.065    216.6      0.0  52.45   4.7    15.055   0.624    0.000  funcE
     2032     4.834(  0.7)     2.379    415.8      0.0  28.61   6.3     4.098   0.246    0.000  funcF
  3147416     0.647(  0.1)     0.000   4232.6   1036.3  47.46   5.0     0.602   0.008    0.000  funcG
        8     0.606(  0.1)    75.725   1739.6     76.4   6.25   5.0     0.061   0.109    0.000  funcH
        4     0.507(  0.1)   126.798   1190.6      0.0  15.07  24.6     0.111   0.001    0.000  funcI
        4     0.376(  0.1)    94.005   3245.3    600.0  54.80  12.1     0.178   0.058    0.000  funcJ
------------------------------------------------------------------------------------------------------
  3524029   685.234(100.0)     0.194   2213.2    232.2   6.54  15.7    76.912  32.748    0.000  total

    62248    37.709(  5.5)     0.606   2200.2   1026.4   0.00   0.0     0.000   0.532    0.000  loop#1
     2032     4.834(  0.7)     2.379    415.8      0.0  28.61   6.3     4.098   0.246    0.000  loop#2
```

関数 ftrace_region_begin と ftrace_region_end を呼んだ時に使った文字列が PROC.NAME の項目に使われ、合計の次の行から範囲ごとの性能情報が表示されます。この情報を基にしてソースコード中の時間がかかっている部分を特定し、プログラム最適化すべき部分を見定めます。同時に、ベクトル演算率などの性能情報を参考にして、どのようにプログラムを修正すれば性能が改善するのかを考えます。ベクトル演算率を高めるためのプログラム最適化については、第 4 章で説明します。

第 3 章のポイント

- ベクトルアーキテクチャである SX-Aurora TSUBASA にとって最重要な性能指標として、ベクトル演算率と平均ベクトル長が挙げられます。
- プログラム全体の性能に関する統計情報を、PROGINF で簡単に調べることができます。実行時に環境変数 VE_PROGINF を YES や DETAIL に設定するだけで、プログラムの実行終了時に性能に関する統計情報が表示されます。
- プログラムのどの部分で、どういう理由で、どれだけ時間がかかっているのかを調べるために FTRACE を使い

ます。そのためには、コンパイル時に-ftrace オプションが必要です。プログラムを実行することで取得された性能情報を、ftrace コマンドを使って表示できます。ソースコード中に関数呼び出しを挿入することで、コード中の任意の範囲の性能情報を取得することもできます。ただし、FTRACE を使う場合には性能が低下しますので、あくまでも性能解析の時にだけ FTRACE を使うようにします。

第 4 章

ベクトルプログラミング

第 1 章の冒頭で述べた通り、コンピュータの設計において

Make the common case fast (よくある処理が速くなるように設計する)

は重要な基本方針です。すなわち、コンピュータには処理の得手不得手が存在します。このため、高速に動作するようにプログラムを修正する、いわゆる**プログラム最適化**においてプログラマがやらなければならないことは、

Make the fast case common (速く実行できる処理を多く使う)

だといえます。つまり、そのコンピュータにとって得意な処理が頻出するようにプログラムを修正することができれば、そのプログラム全体としても高速化が期待できるはずです。

第 1 章で述べた通り、SX-Aurora TSUBASA にとっての最も重要な fast case は、長いベクトルに対してベクトル命令を実行している場合です。本章では、なるべく多くのループをベクトル化し、ベクトル長も増やすためのプログラム最適化技法について説明します。

4.1 ループのベクトル化

SX-Aurora TSUBASA の性能を引き出すために最も重要なことは、なるべく多くの処理をベクトル命令を使って行うということです。すなわち、第 3 章で紹介した**ベクトル演算率**を高めることです。一般的なプログラマは C や Fortran 等のいわゆる高水準プログラミング言語を使ってプログラムを書くため、ベクトル命令を使うか否かを実際に判断しているのはプログラマではなく、プログラムを機械語命令に翻訳する**コンパイラ**です。したがって、できるだけベクトル命令を使って処理を行うということは、できるだけ多くの処理がベクトル命令へと翻訳してもらえるように、コンパイラに配慮してプログラムを書くことと言い換えることができます。

プログラム最適化においてプログラマは、第 3 章で述べたようにプログラム中で実行時間の長い部分を見つけ、その部分を優先的に高速化します。多くの場合、特に科学技術計算の場合には、実行時間の長い部分がループで記述されています。「ループを**ベクトル化**する」とは、そのループがベクトル命令を使った機械語に変換されることを意味しています。本文書では、ベクトル化してもらえるようにコンパイラに配慮しながらプログラミングすることを、**ベクトルプログラミング**と呼びます。

配列 A と配列 B に格納されたベクトルの加算を例にして、ベクトル命令による演算の概念を図 4.1、図 4.2、図 4.3 で説明します。まず、図 4.1 では配列要素を 1 つずつ加算しています。このような実行方法を**逐次実行**といいます。この場合、ハードウェア構成をシンプルにできるので 1 回の演算に要する時間 (**遅延時間**あるいは**レイテンシ**) は短くて済むかもしれませんが、配列要素がたくさんある場合には全体の実行時間は長くなってしまいます。図 4.2 では、ある配列要素の加算をしている途中で他の配列要素の加算も始めています。このように複数の配列要素の加算を時間的にオー

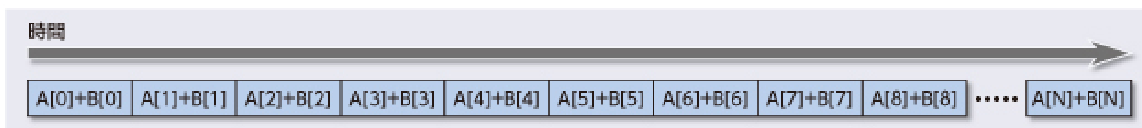


図 4.1 ベクトル同士の加算の逐次実行

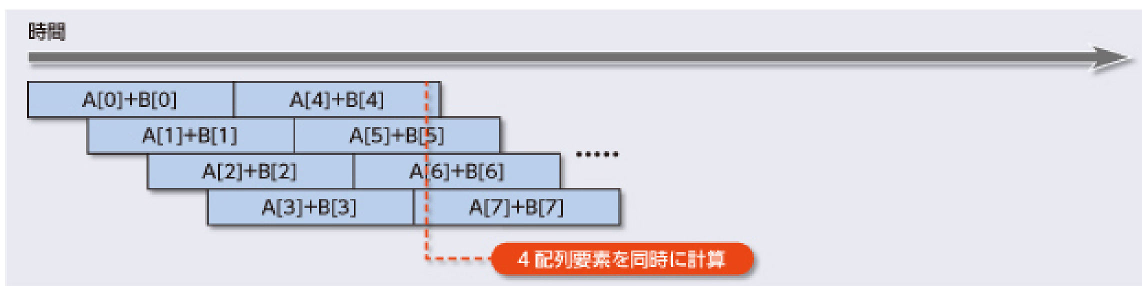


図 4.2 ベクトル同士の加算のパイプライン実行

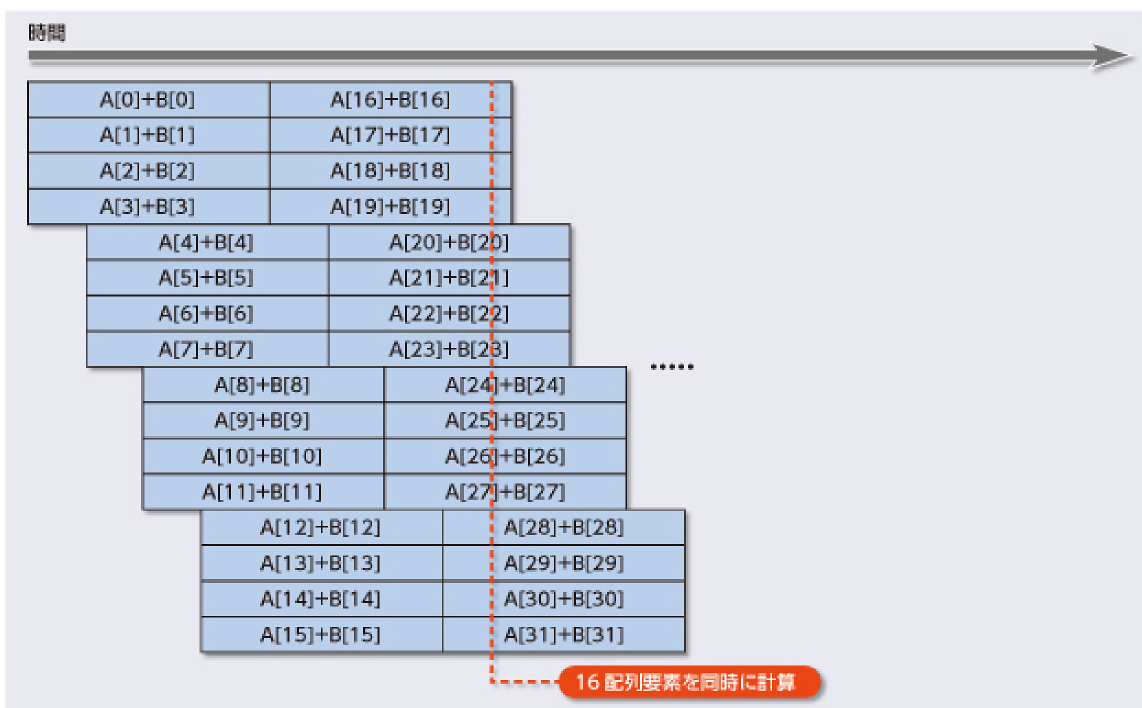


図 4.3 ベクトル同士の加算のベクトル実行 (実際の SX-Aurora TSUBASA の並列数は図の 8 倍)

バーラップすることで、全体の実行時間を短くしています。このような実行方法を**パイプライン実行**と呼びます。古典的なベクトルプロセッサはベクトル演算をパイプライン実行することで効率的な処理を実現してきました。SX-Aurora TSUBASA の**ベクトル演算器**には FMA 演算器が 32 基搭載されていますので、複数の演算要素を並列計算しつつパイプライン実行して、さらに効率のよいベクトル演算を行っています。図 4.3 では、4 基の演算器が並列動作していると仮定した場合のベクトル処理を示しています。ある時点で計算されている配列要素数を見ると、図 4.2 では最大で 4 つの配列要素が計算されているのに対して、図 4.3 では 32 もの配列要素が計算されていることがわかります。図の都合

上、図 4.3 では 4 つの演算器の並列動作の様子が描かれていますが、実際の SX-Aurora TSUBASA では 32 基の演算器が並列に動作しますので、256 もの配列要素が計算されている状態になります。パイプライン実行やベクトル実行を行うためにハードウェア構成が複雑になり、1 回の演算に要する時間は長くなるかもしれませんが、単位時間あたりに実行できる演算回数 (**スループット**) は大幅に向上することがわかります。一般的に、大規模な科学技術計算では膨大な数のデータを処理しなければならないため、スループットが高いということは科学技術計算アプリケーションの高速化に非常に有効です。

図 4.2 や図 4.3 から、同時に計算されている配列要素が最大値に達するまでにある程度の時間を要することもわかります。計算が始まった直後には、まだ少ない数の配列要素しか計算中になりませんが、徐々に計算中の配列要素数が増えていきます。図 4.2 の場合には 4 配列要素、図 4.3 の場合には 32 配列要素が計算中になる効率的な状況は、配列要素数が多ければ多いほど長く続きます。配列要素数が少なければ、効率の良い状況はすぐに終わってしまいますし、極端に少ない場合には図 4.1 の逐次実行の方が早く終わるということもあり得ます。このことから、ベクトル長をなるべく増やした方が、効率的なベクトル処理を期待できることがわかります。

古典的なベクトルプロセッサでは、ベクトル演算率とベクトル長を増やすことだけがベクトルプログラミングの目標でした。しかし、近年ではベクトル長を増やせば性能が上がるとは一概には言えなくなってきました。現在のベクトルプロセッサには図 1.7 のようにキャッシュが搭載されており、キャッシュ上のデータには高速にアクセスできるようになっています。ベクトル長を増加させることでデータアクセスの局所性が低下し、キャッシュミス率も増える傾向がありますので、ベクトル長の増加によるベクトル演算効率化とベクトル長の減少によるキャッシュヒット率向上 (メモリアクセス性能向上) にはトレードオフが存在することがあります [7]。これらの両方の要因を考えながら、性能が高くなるようにプログラムを修正していく作業が現代風のベクトルプログラミングといえるでしょう。

4.2　コンパイルリスト

第 2 章のコンパイルオプションの説明にもあるように、SX-Aurora TSUBASA の VE 向け C/C++ コンパイラでは`-report-all` オプションをつけることで、**診断メッセージリスト**と**編集リスト**を表示します。診断メッセージリストと編集リストの総称として、**コンパイルリスト**という用語もあります。これらのリストには、コンパイラからプログラマに向けたメッセージが書かれています。

例えば、リスト 2.1 を`-O3 -report-all` をつけてコンパイルすると、診断メッセージリストと編集リストがまとめられ 1 つのファイルとして出力されます。ファイル名は拡張子`.L` になります。

Listing 4.1　診断メッセージリストと編集リストを同時出力した例

```
NEC C/C++ Compiler (1.3.0) for Vector Engine     Fri Aug 24 16:54:27 2018
FILE NAME: sample.c

FUNCTION NAME: main
DIAGNOSTIC LIST

 LINE              DIAGNOSTIC MESSAGE

    15: vec( 101): Vectorized loop.
    19: opt(1600): Array delinearized.: a
    24: opt(1589): Outer loop moved inside inner loop(s).: j
    24: vec( 101): Vectorized loop.
    26: opt(1592): Outer loop unrolled inside inner loop.: i
    26: vec( 101): Vectorized loop.
    27: vec( 128): Fused multiply-add operation applied.
```

```
NEC C/C++ Compiler (1.3.0) for Vector Engine     Fri Aug 24 16:54:27 2018
FILE NAME: sample.c

FUNCTION NAME: main
FORMAT LIST

 LINE   LOOP      STATEMENT

     5:           int main(int argc, char** argv) {
     6:             int i,j,k;
     7:             double *a, *b, *c;
     8:
     9:             a = (double*)malloc(N*N*sizeof(double));
    10:             b = (double*)malloc(N*N*sizeof(double));
    11:             c = (double*)malloc(N*N*sizeof(double));
    12:
    13:             /* 行列の初期化 */
    14: +------>    for(j=0;j<N;j++){
    15: |V----->      for(i=0;i<N;i++){
    16: ||              a[j*N+i]=i*j;
    17: ||              b[j*N+i]=(i==j?1:0);
    18: ||              c[j*N+i]=0.0;
    19: |V-----       }
    20: +------     }
    21:
    22:             /* 行列積の計算 */
    23: +------>    for(k=0;k<N;k++){
    24: |X----->      for(j=0;j<N;j++){
    25: ||              c[k*N+j] = 0.0;
    26: ||V---->        for(i=0;i<N;i++){
    27: |||     F         c[k*N+j] += a[k*N+i]*b[i*N+j];
    28: ||V----         }
    29: |X-----       }
    30: +------     }
    31:
    32:             return 0;
    33:           }
```

リスト 4.1 からもわかるように、前半の診断メッセージリストでは左側から順に行番号とそれに対するコンパイラメッセージが表示されており、後半の編集リストでは左側から順に行番号、最適化情報、元コードが表示されています。また、リスト 4.1 では関数 main の診断メッセージリストと編集リストのみが出力されていますが、プログラムの中に複数の関数が定義されている場合には、コンパイルリストが関数ごとに出力されます。

最適化情報には以下のような意味があります。

ループ全体がベクトル化された

```
V----->    for(i=0;i<N;i++){
```

```
|               ...
V-----      }
```

ループがベクトル化されなかった

```
+----->     for(i=0;i<N;i++){
|               ...
+-----      }
```

ループが部分的にベクトル化された (部分ベクトル化)

```
S----->     for(i=0;i<N;i++){
|               ...
S-----      }
```

ループが並列化された

```
P----->     for(i=0;i<N;i++){
|               ...
P-----      }
```

ループがベクトル化され、かつ並列化された

```
Y----->     for(i=0;i<N;i++){
|               ...
Y-----      }
```

2 つのループが一重化 (collapse) された

```
W----->     for(i=0;i<N;i++){
|*---->       for(j=0;j<N;j++){
||                ...
|*----        }
W-----      }
```

外側と内側のループが交換 (interchange) された

```
X----->    for(i=0;i<N;i++){
|*---->      for(j=0;j<N;j++){
||             ...
|*----       }
X-----     }
```

外側ループが外側アンロール (アウターアンロール) されてから、内側ループがベクトル化された

```
U----->    for(i=0;i<N;i++){
|V---->      for(j=0;j<N;j++){
||             ...
|V----       }
U-----     }
```

2 つのループが融合 (fusion) されてベクトル化された

```
V----->    for(i=0;i<N;i++){
|            ...
|          }
|          for(i=0;i<N;i++){
|            ...
V-----     }
```

ループが展開 (expand) された (この例ではループ本体の内容を 4 回繰り返して書くことで、ループがなくなった)

```
*----->    for(i=0;i<4;i++){
|            ...
*-----     }
```

編集リストの中では、ループではない行に最適化情報が表示されることもあります。例えばリスト 4.1 の編集リストでは、27 行目の文に対して F という最適化情報が表示されています。このように、ループ以外の行に対する最適化情報は一文字で表されています。主な最適化情報の意味は以下のとおりです。

`I`
関数呼び出しがインライン化された。
`M`
多重ループ全体がライブラリ関数呼び出しに置き換えられた。
`F`
式に対してベクトル積和演算命令が使われた。
`G`
ベクトル収集 (ギャザー) 命令が使われた。
`C`
ベクトル拡散 (スキャッタ) 命令が使われた。
`R`
配列ができるだけキャッシュ上に保持されるように指示された (`retain` 指示行に相当)。
`V`
配列ができるだけレジスタ上に保持されるように指示された (`vreg` 指示行に相当)。

ベクトルプログラミングの際には、診断メッセージリストや編集リストに書かれたコンパイラからのメッセージに耳を傾けながら、プログラムを修正する作業を進めていきます。

4.3 ベクトル化の促進

SX-Aurora TSUBASA の VE 向けのコンパイラは**自動ベクトル化機能**を持っていますので、できるだけ多くのループをベクトル化しようと試みます。しかし、「ループをベクトル化してもプログラムの挙動が変わらない」ことをコンパイラが判断できない場合、そのループはベクトル化されません。また、性能向上が期待できない場合にも、ループはベクトル化されません。したがって、ベクトルプログラミングの主な要点は以下の 2 つです。

1. ベクトル化できるループである (ベクトル化がプログラムの挙動を変えない)、ということをコンパイラが判断しやすいように配慮する。コンパイラはベクトル化のための条件に合ったループのみを実際にベクトル化するため、その条件を満たすように考えてループを書く。
2. 図 3.2 でも説明した通り、ベクトル命令の実行にはある程度の立ち上がり時間が必要なため、すぐに処理が終わってしまうような短いベクトルでは、立ち上がり時間の分だけ実行時間が伸びてしまうかもしれない。このため、ベクトル化による性能向上が期待できるように、できるだけベクトル長を増加させる。

ベクトル化のための条件は大きく分けて以下の 5 種類です。

- ベクトル化可能なループ構造であること
- ループ制御変数とループ終了式がベクトル化に適していること
- ループ本体がベクトル化の対象となる文のみを含んでいること
- ループ本体がベクトル化の対象となる演算子のみを含んでいること
- ループ本体がベクトル化を妨げるデータの依存関係を含まないこと

以上の条件を満たさなくなる要因を、**ベクトル化阻害要因**と呼びます。以下、SX-Aurora TSUBASA の VE 向けの NEC 社製 C/C++ コンパイラにおける、ベクトル化の条件とベクトル化阻害要因の例を説明します。

ただし、ベクトル化の条件を厳密には満たさない場合でも、よく使われる特別なパターンに関してはコンパイラが専

用のベクトル命令を用いてベクトル化することができます。そのような特別なパターンは**マクロ演算**と呼ばれ、いくつかの種類があります。マクロ演算に関しては第 4.3.5 節で説明します。

4.3.1 ループ構造

SX-Aurora TSUBASA 向けの C/C++ コンパイラにおいて、ベクトル化可能なループは `while` 文、`do` 文、および `for` 文で書かれたループ構文です。

Listing 4.2 `while` 文によるループの例

```
i=0;
while(i<N){
  /* ループ本体 */
  i++;
}
```

Listing 4.3 `do` 文によるループの例

```
i=0;
do {
  /* ループ本体 */
  i++;
} while(i<=N);
```

Listing 4.4 `for` 文によるループの例

```
for(i=0;i<N;i++) {
  /* ループ本体 */
}
```

ループが入れ子状になっている場合、最も内側のループ (**最内ループ**) が最初にベクトル化の対象として考えられます。以下の例では、変数 `j` で制御されている内側のループがまずはベクトル化される候補になります。

Listing 4.5 入れ子状になった多重ループ (**ループネスト**) の例

```
for(i=0;i<N;i++) {
  for(j=0;j<M;j++){
    /* ループ本体 */
  }
}
```

ただし、外側のループと内側のループが交換可能で、入れ替えたほうが高い性能を期待できるとコンパイラが判断した場合、コンパイラが自動的に**ループ交換** (loop interchange) を行う場合もあります。また、最内ループが短いことが自明である場合には、最内ループを自動的に**ループ展開** (loop expansion) して外側のループでベクトル化を行う場合もあります。

ベクトル化のためには、ループ本体への入口と出口がそれぞれ 1 つずつである必要があります。原則的に、入口や出口が複数ある場合にはベクトル化の対象にはなりません (ただし、第 4.3.5 節で後述する**マクロ演算**の条件を満たす場合にはベクトル化されます)。

Listing 4.6 入口が複数あるループの例

```
if(k<0){
```

```
  goto label1;
}
for(i=0;i<N;i++) {
label1:
  /* ループ本体 */
}
```

Listing 4.7　出口が複数あるループの例 1

```
for(i=0;i<N && j<0;i++) {
  /* ループ本体 */
}
```

リスト 4.6 では、goto 文でループ本体に入る場合と、通常どおり for 文を実行してループ本体に入る場合があります。このように入口が複数ある場合、ループはベクトル化されません。同様に、リスト 4.7 では、2 つの条件のうちいずれかを満たさなくなった場合にループ本体を出るため、それぞれの条件に対応した 2 つの出口があると考える必要があります。これらのように出口が複数ある場合、通常、ループはベクトル化されません。ベクトル化できないループ構造だとコンパイラが判断した場合には、阻害要因の位置 (行番号) とともに `Unvectorized loop structure.` というメッセージが表示されます。そのような場合、ベクトル化のためには、それぞれ 1 つずつになるようにプログラムを修正する必要があります。

Listing 4.8　出口が複数あるループの例 2

```
for(i=0;i<N;i++) {
  /* ループ本体 */
  if(j>=0) break;
}
```

リスト 4.8 のループでも、処理を N 回繰り返してからループ本体を出る場合と、break 文でループ本体から出る場合があります。すなわち、2 つの出口があります。しかし、リスト 4.8 は**サーチ**という種類のマクロ演算 (第 4.3.5 節参照) と認識され、ベクトル化されます。

4.3.2　ループ制御変数

ループの繰り返しにしたがって、ある決まった増分で増加する変数を**インダクション変数**と呼びます。例えば、ループが繰り返すたびに i++ や i--で増加/減少する変数 i は、インダクション変数です。インダクション変数の値を計算する式を、**増分式**と呼びます。増分式が i=i+C のとき、C が**ループ内不変式** (ループを繰り返している間に値が変化しない) であれば変数 i はインダクション変数であり、C はその**増分値**と呼ばれます。

インダクション変数のうち、ループの終了条件に関わる変数を**ループ制御変数**と呼びます。コンパイラがベクトル化を行うためには、ループの制御変数と終了条件が以下の 2 つの条件を満たしている必要があります。

1. ループ制御変数が 4 バイト以上の整数型[*1]のスカラ変数であること。
2. ループ終了式が以下の不等式と等価であること。
   - 増分値が正のとき (ループの繰り返しによってループ制御変数が増加しているとき)

$$\mathtt{i} < expr \quad \text{あるいは} \quad \mathtt{i} <= expr$$

[*1] 具体的な変数型として int、unsigned int、long、unsigned long、long int、unsigned long int、long long、unsigned long long、long long int、unsigned long long int が該当します。

- 増分値が負のとき (ループの繰り返しによってループ制御変数が減少しているとき)

  $$\mathtt{i} > expr \quad \text{あるいは} \quad \mathtt{i} >= expr$$

  ただし、ここで $expr$ はループ内不変式であり、その式の中に関数呼び出しがない。

ループ制御変数の型やループ終了条件が条件を満たしていない場合、ベクトル化のためにそれらを変更する必要があります。ループ制御変数や終了条件が理由でベクトル化できないとコンパイラが判断した場合には、阻害要因の位置 (行番号) とともに `Unvectorized loop structure.` というメッセージが表示されます。

4.3.3 ループ本体内の構成要素

ループをベクトル化するためには、ループ本体を構成する要素に制限があります。
まず、ベクトル化の対象となるためには、ループ本体が以下の文のみを含んでいる必要があります。

- if 文
- switch 文
- goto 文
- continue 文
- break 文
- 式文 (演算式のみで構成される文)

文にラベルがついている場合にも、ベクトル化阻害要因にはなりません。

```
label1: a[i] = a[i] + b[i];
```

また、複合文 (ブロック) になっていても、ベクトル化阻害要因にはなりません。

```
if(i>10)
/* 複数の文から構成される複合文 */
{
  a[i] = a[i] + b[i];
  a[i] *= 10;
}
```

ここで注目したいのは、関数呼び出しはベクトル化阻害要因に「なる」ということです。このため、ループ本体に関数呼び出しが含まれている場合、ベクトル化するためには**インライン展開**が必要です。例えば、リスト 4.9 の 2 番目のループのようにループ本体を直接書き換えて関数呼び出しを回避することも可能です。また、コンパイラにも関数のインライン展開を自動的に行う機能があります。コンパイラオプション`-finline-functions` を指定することで、コンパイラが可能だと判断した関数呼び出しをインライン展開します。

Listing 4.9 手動インライン展開

```
/* ユーザ定義関数 */
int add(int a, int b) {return a+b;}
...
/* 関数の呼び出しが阻害要因でベクトル化されない */
for(i=0;i<100;i++){
  z[i] = add(x[i],y[i]);
}
```

```
  /* 阻害要因がないためにベクトル化される */
 for(i=0;i<100;i++){
   /* 関数内の処理をループ本体内に直接書くことで関数呼び出しを回避 */
   z[i] = x[i]+y[i];
 }
```

関数呼び出しが阻害要因となってベクトル化できなかった場合には、その関数呼び出しの位置 (行番号) とともに `Vectorization obstructive function reference.:` *FUNC* というメッセージが表示されます。ここで *FUNC* は呼び出されている関数名が表示されます。また、コンパイラがインライン展開を行った場合には、その行番号とともに `Inlined:` *FUNC* と表示されます。また、編集リストでも関数を呼び出している行に I マークがつきます。

4.3.4 依存関係

複数の処理に関して実行順序が決められており、その順序を守らないと実行結果が変わってしまう場合、「それらの処理の間には**依存関係**がある。」と表現されます。図 4.3 からもわかるように、ループをベクトル化することで演算やメモリアクセスのタイミングが変化し、その結果として実行順序が厳密には逐次実行とは異なります。このため、ループにベクトル化を阻害する依存関係がある場合、あるいはあるかどうかを判断できない場合には、コンパイラはそのループをベクトル化しなかったり、部分的にのみベクトル化したりします。したがって、ループの依存関係を意識してなるべくベクトル化されるようにループ構造等を修正することは、ベクトルプログラミングの中でも特に重要なポイントです。

コンパイラがループを解析し、阻害要因となるような依存関係がないと判断した場合のみ、そのループはベクトル化されます。例えば、リスト 4.10 のループの場合 i 番目の繰り返しのときに 2 行目で定義された配列要素が、$i+1$ 番目の繰り返しのときに 3 行目で参照されています。必ず、定義された値が時間的に後から参照される、という順序が決まっています。

Listing 4.10 ベクトル化可能なループ (分割前)

```
1 sum=0;
2 for(i=1;i<N;i++){
3   a[i]= i;
4   sum += a[i-1];
5 }
```

つまりこの例では、2 行目で上書きされる古い値が、3 行目で参照されることはありません。このため、リスト 4.11 のように 2 行目の実行を任意の順番で繰り返してから、3 行目の実行を任意の順番で繰り返すことができます。

Listing 4.11 ベクトル化可能なループ (分割後)

```
1 sum=0;
2 for(i=1;i<N:i++)
3   a[i]= i;
4 for(i=1;i<N:i++)
5   sum +=a[i-1];
```

ループ分割前後での実行順序の違いを図 4.4 に示します。図中の青い処理をまとめて実行し、その後にオレンジ色の処理をまとめて実行しても結果が変わらないことがわかります。このためこの例の場合、ベクトル化阻害要因となる依存関係はありません。依存関係の有無が判断しづらい場合には、リスト 4.11 のようにループを分割して、1 行ずつループで繰り返すことを頭の中で考えてみるのが一案です。ループを分割すると実行順序が変わりますが、それでも結果が変わらない場合には依存関係がないと言えます。

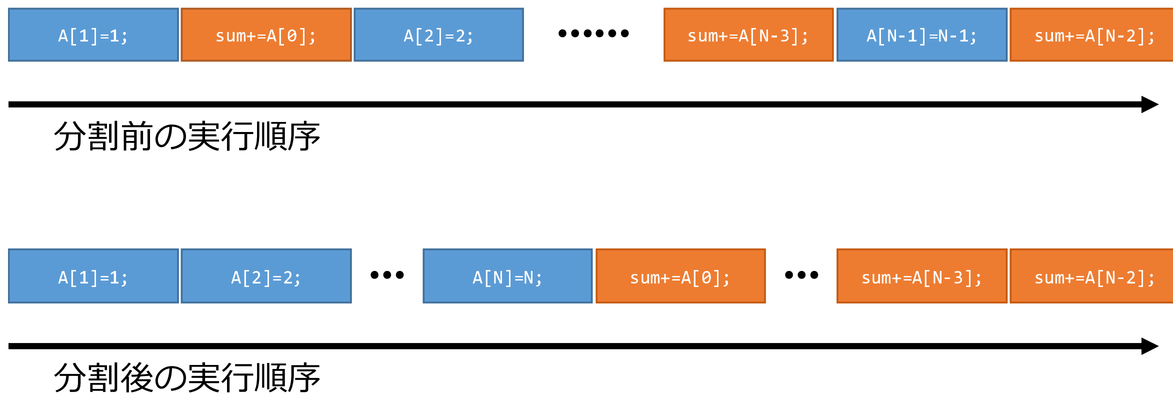


図 4.4　ループ分割による実行順序の変化 (計算結果は同じ)

定義と参照が同じ行にある場合、仮の配列 (リスト 4.12 の例での配列 c) を使って行を分ければ、ループ分割によって挙動が変わるかどうかを確認できます。

Listing 4.12　定義と参照が同じ行にある場合の考え方

```
/* 分割前 */
for(i=1;i<N;i++)
  a[i-1] = a[i] + b[i];

/* 分割後 */
for(i=1;i<N;i++)
  c[i] = a[i] + b[i];
for(i=1;i<N;i++)
  a[i-1] = c[i]
```

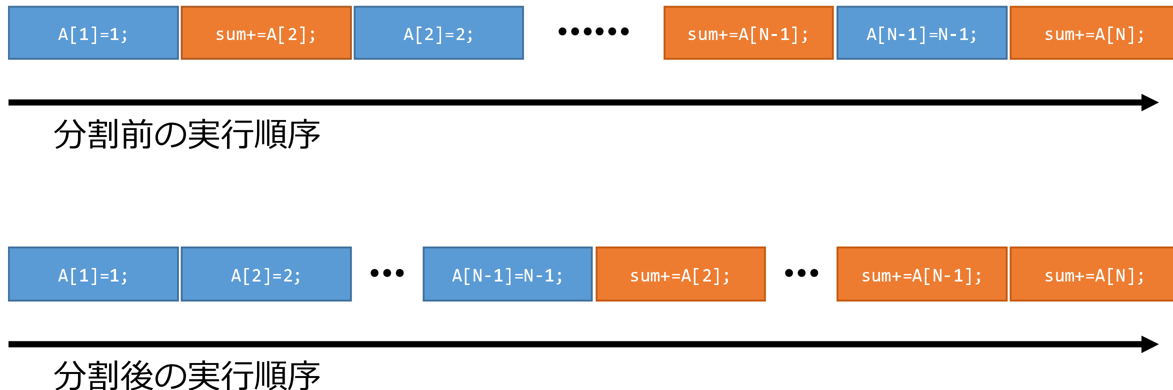
例えば、リスト 4.10 の 3 行目の参照を a[i+1] に書き換えるとループを分割した場合に結果が変わります。参照で、更新される前の古い値が必要になるためです。

Listing 4.13　分割前後で動作が変わってしまう例

```
/* 分割前 */
 sum=0;
 for(i=1;i<N:i++){
   a[i]= i;
   sum +=a[i+1];
 }

/* 分割後 */
 for(i=1;i<N:i++)
   a[i]= i;
 for(i=1;i<N:i++)
   sum +=a[i+1];
```

実行順序の変化を図 4.5 に示します。例えば a[2] に着目すると、分割前は代入前の値が変数 sum の計算に使われています。一方、分割後には a[2]=2 が実行された後に sum の計算で参照されています。このため、計算結果が変化することが分かります。



「値を代入する前の合計」を計算するという
順序が守られていないので計算結果が変わる

図 4.5　ループ分割による実行順序の変化 (計算結果が変化)

リスト 4.14 のループも、分割すると順序が変わり、計算結果が変化してしまいます。つまり、ベクトル化を阻害する依存関係があり、コンパイラはそれを正しく判断してベクトル化しません。

Listing 4.14　ベクトル化を阻害する依存関係のあるループ

```
for(i=0;i<N;i++)
  a[i+1] = a[i] + b[i];
```

このループでは、まずはじめに a[0] を参照して a[1] を定義します。次に、定義された後のa[1] を参照して a[2] を定義します。各配列要素を一つずつ順序通りに定義する必要があり、依存関係があることがわかります。この依存関係のため、一般的に a[i] を定義した後でなければ、a[i+1] を定義することができません。このループが仮にベクトル化された場合、図 4.3 に示す通り複数の配列要素を同時に計算することになりますので、図 4.1 の場合とは結果が変わってしまいます。

ただし、例えばループ変数のような**インダクション変数** (第 4.3.2 節参照) やマクロ演算 (第 4.3.5) は例外です。インダクション変数は、明らかに前の繰り返しで定義された値を使って再定義される変数ですが、コンパイラはそれがインダクション変数であるということを検出して問題がないように対処します。逆に言えば、その検出を妨げるような処理が含まれている場合にはベクトル化が阻害されてしまうかもしれないので、注意が必要です。例えば、インダクション変数がループ本体内で型変換されている場合などには、ベクトル化が阻害されることがあります。ループ本体内でインダクション変数の値を操作するのは、避けたほうが無難です。

依存関係の有無は、プログラムを見ただけでは完璧には判断できません。リスト 4.15 のループの場合、k の値によってはベクトル化を阻害する依存関係があるかもしれませんが、ないかもしれません。具体的には、変数 k が 0 以下の値ならば、ベクトル化を阻害する依存関係はありません。

Listing 4.15　ベクトル化を阻害する依存関係があるかもしれないループ

```
for(i=1;i<N;i++)
```

```
  a[i+k] = a[i] + b[i];
```

リスト 4.15 のループで k=-1 の場合を考えましょう。まず最初の繰り返しでは、右辺で a[1] を使って左辺の a[0] が定義されます。次の繰り返しでは、右辺で a[2] を使って左辺の a[1] が定義されます。このようにして、右辺に出てくる値はこのループ本体で計算される前の値になります。このため、前の繰り返しの結果を待たずに配列要素を計算できるわけです。この場合、実際にはベクトル化しても結果は変わりませんが、コンパイラは k が 0 以下の値になることを確信できません。このため、このままではループはベクトル化されません。コンパイラは Dependency unknown. Unvectorizable dependency is assumed: *VAR* というメッセージを表示します。ここで *VAR* はベクトル化阻害要因となった変数の名前です。

変数 k が確実に 0 以下になる場合、プログラマはリスト 4.15 のループにベクトル化を阻害する依存関係がないと判断できます。その判断結果をコンパイラに教えてあげることで、このループをベクトル化することができます。そのようにコンパイラに対して付加情報を伝えるために**コンパイラ指示行** (compiler directive) があります。コンパイラ指示行は#pragma _NEC *Keyword* という書式になっており、ループに依存関係がないことを指示するためには、*Keyword* として ivdep を使います。以下の例の場合、ループには依存関係がないと仮定してベクトル化が行われます。

Listing 4.16　コンパイラに依存関係がないことを指示する方法

```
#pragma _NEC ivdep
for(i=-k;i<N;i++)
  a[i+k] = a[i] + b[i];
```

ここまでに登場したいくつかのプログラム例からもわかるように、コンパイラが依存関係の有無を確認するためには、配列要素へのアクセスを適切に解析しなければなりません。アクセスされる配列要素は、配列の**添字** (**インデックス**) によって決まりますので、添字計算の式が複雑な場合には依存関係がコンパイラにとってわかりづらくなります。多くの場合、添字計算が線形式の場合にはコンパイラは依存関係を判断しやすいといえます。すなわち、リスト 4.17 の配列 c のように連続する配列要素を一つずつアクセス (**シーケンシャルアクセス**) や、配列 b のように一定間隔 (**ストライド**) で配列にアクセス (**ストライドアクセス**) する場合などです。一方、配列 a のように、別の配列 d の要素を添字として使って配列にアクセス (**リストアクセス**) する場合、コンパイラは添字がどのような値を取るのかを知ることができません。この場合、コンパイラは依存関係を解析できません。例えば、配列 d の複数の要素が同じ値を保持している場合、実行の順番を変えると結果も変わってしまいますが、配列 d の値をコンパイル時に知ることはできません。このため、コンパイラ指示行などで依存関係の有無を指示しない限り、リスト 4.17 のループはベクトル化されません。

Listing 4.17　リストアクセス

```
for(i=0;i<N;i++)
  a[d[i]] = b[2*i+1] + c[i];
```

配列ばかりではなく、通常のスカラ変数がベクトル化阻害要因となる依存関係を持つこともあります。リスト 4.18 の例の場合、i 番目の繰り返しで定義された s の値が i+1 番目の繰り返しで参照されています。このように、スカラ変数の定義の前に参照がある場合、ベクトル化されません。

Listing 4.18　スカラ変数によるベクトル化阻害

```
for(i=0;i<N;i++){
   a[i] = s;
   s = ...;
}
```

また、リスト 4.19 の例では `if` 文の本体が実行された場合のみ変数 `s` の値が再定義され、それ以外の場合には定義されません。このため、`i` 番目の繰り返しで定義された値が、`i+1` 以降の繰り返しで使われる可能性を排除できません。このような場合にも、コンパイラはこのループをベクトル化しません。

Listing 4.19　条件分岐によるベクトル化阻害

```
for(i=0;i<N;i++){
  if( cond ) {
    s = ...;
  }
}
```

スカラ変数がベクトル阻害要因となる依存関係を持たないために、スカラ変数の値が後続の繰り返しでは使われないことをできるだけ明確にコンパイラに教えてあげるべきです。そのためには、ループ本体の冒頭で変数に初期値を与える、あるいは条件分岐先に関係なく必ず変数に値が定義されるように修正する、などの選択肢があります。

Listing 4.20　明示的な定義によるベクトル化

```
for(i=0;i<N;i++){
  s = 0; /* 初期化により、以前の繰り返しで定義された値を破棄 */
  if( cond ) {
    s = ...;
  }
  else {
    /* 条件分岐結果に関係なく必ず定義されるようにする */
    s = ...;
  }
}
```

以上のような配慮をすることによって、科学技術計算で出てくる多くのループ構造はベクトル化可能になるでしょう。ベクトルプログラミングは長い歴史を持つため、そのままではベクトル化できないループ構造に関しても、ベクトル化するためのさまざまな技法が先人たちによって考えられています。そのような技法の例に関しては、第 7 章で紹介します。

4.3.5　マクロ演算

これまでに説明してきたように、原則的に、コンパイラはベクトル化の条件を満足するループをベクトル化します。しかし、ベクトル化の条件を厳密には満たさない場合でも、よく使われる特別なパターンに関してはコンパイラが専用のベクトル命令を用いてベクトル化することができます。そのような特別なパターンを**マクロ演算** [1] と呼びます。最適化オプションが`-O1` 以上のときにマクロ演算機能が有効になります。

例えば、以下のコード例にはイテレーション間に依存関係がありますが、ベクトルコンパイラはマクロ演算であることを認識してベクトル化します。

Listing 4.21　リダクションの例

```
for(i=0;i<N;i++){
  s = s + exp;
}
```

Listing 4.22　漸化式の例

```
for(i=0;i<N;i++){
  a[i] = exp1*a[i-1] + exp2;
}
```

Listing 4.23　最大値/最小値の例

```
for(i=0;i<N;i++){
  if(amax < a[i]){
    amax = a[i];
  }
}
```

Listing 4.24　サーチの例

```
for(i=0;i<N;i++){
  if(a[i]==0){
    break;
  }
}
```

Listing 4.25　圧縮/伸長の例

```
j=0;
for(i=0;i<N;i++){
  if(a[i]>=0){
    b[j] = a[i];
    j=j+1:
  }
}
```

なお、マクロ演算のサーチと認識されるための条件は以下の通りです。

- 最内側のループである。
- ループの外への分岐はただ一つである。
- ループの外への分岐条件はループの繰返しに依存する。
- ループの外への分岐の前に配列要素、ポインタの指示先への代入がない。
- ループの外への分岐以外は、ベクトル化の条件をすべて満たしている。

4.4　ベクトル長の増加

SX-Aurora TSUBASA は、効率のよくベクトル命令を実行することを最優先に考えて設計されています。いったんベクトル処理が始まってしまえば、図 4.3 で示したように多数の演算が同時進行で実行 (**並行実行**) され、高いスループットを達成できます。しかし、図 3.2 に示すように、効率的なベクトル処理が始まるまでにある程度の準備時間 (**立ち上がり時間**) を要しますので、ベクトル処理がすぐに終わってしまうような場合には却って性能が低下してしまうこともあります。効率的なベクトル処理を行うためには、ある程度のベクトル長が必要です。ループがベクトル化されますので、プログラマはベクトル化の対象となるループの長さ (**ループ長**) を増やす必要があると言い換えることができます。

ベクトル長と性能との関係の典型例を図 4.6 に示します。縦軸は図中に示された `for` ループ (配列の積和演算) を実行したときの性能、横軸はベクトル化されるループの長さ (すなわちベクトル長) を表しています。この図からもわか

るように、ベクトル長が短いときには効率的なベクトル処理を行うことができず、結果として低い性能しか達成できていません。ベクトル長が延びるにつれて性能が向上している様子を見ることができます。

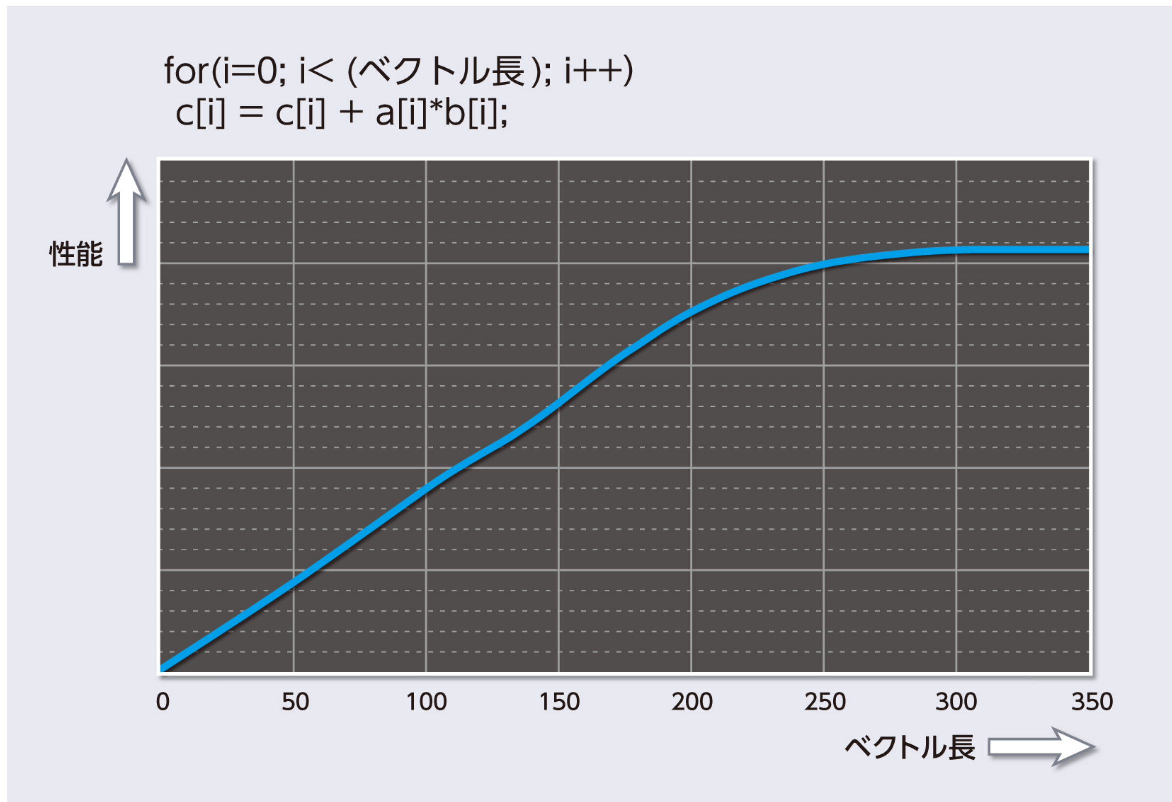


図 4.6　典型的なベクトル長と性能との関係

ループ長を伸ばすために最もよく使われる一般的な技法として、**ループ一重化** (loop collapse) があります。

Listing 4.26　ループ一重化の例

```
/* 一重化前 */
for(i=0;i<N;i++){
  for(j=0;j<M;j++){
    a[i][j] = ...;
  }
}

/* 一重化後 */
for(ij=0;ij<N*M;ij++){
  i=ij/M;
  j=ij%M;

  a[i][j] = ...;
}
```

ほとんどの場合、科学技術計算で実行時間のかかるループは多重ループ (ループネスト) になっています。入れ子になっている個々のループは短い場合でも、それらをまとめて一重ループにしてしまえば、ループの長さを増やすことができるというメリットがあります。ただし、リスト 4.26 の一重化後のループのように、プログラムの挙動を変えないため

にはループ本体で配列添字用に変数の再計算が必要になることがあり、その分だけ余計な計算が必要になるというデメリットもあります。また、多次元配列の**袖領域** (図 4.7 参照) の配列要素だけ、他の内側の配列要素とは別の処理をしなければならないことがありますが、そのような場合にループを一重化してしまうと、余計な計算が増えたり、プログラムが複雑になったりするデメリットが生じることもあります。一般的に、ループ本体内での計算が十分に多い場合には、メリットのほうが大きくなります。このため、ループ本体内での計算を多くして配列添字用変数計算などのコストを相対的に低くし、また計算後の値の再利用率を高めるために**ループ融合** (loop fusion) の適用も検討する価値があります。

Listing 4.27　ループ融合の例

```
/* 融合前 */
for(ij=0;ij<N*M;ij++){
  i=ij/M;
  j=ij%M;

  a[i][j] = ...;
}
for(ij=0;ij<N*M;ij++){
  i=ij/M;
  j=ij%M;

  b[i][j] = ...;
}


/* 融合後 */
for(ij=0;ij<N*M;ij++){
  i=ij/M;
  j=ij%M;

  /* つのループでの計算を一つにまとめた2 */
  a[i][j] = ...;
  b[i][j] = ...;
}
```

多重ループの外側のループが長い場合、**ループ交換** (loop interchange) によってベクトル長を増やせることもあります。リスト 4.28 では最内ループの長さが M から N に変化していますので、後者の方が長い場合にはベクトル長の増加を期待できます。

Listing 4.28　ループ交換の例

```
/* ループ交換前 */
for(i=0;i<N;i++){
  for(j=0;j<M;j++){
    a[i][j] = ...;
  }
}

/* ループ交換後 */
for(j=0;j<M;j++){
  for(i=0;i<N;i++){
```

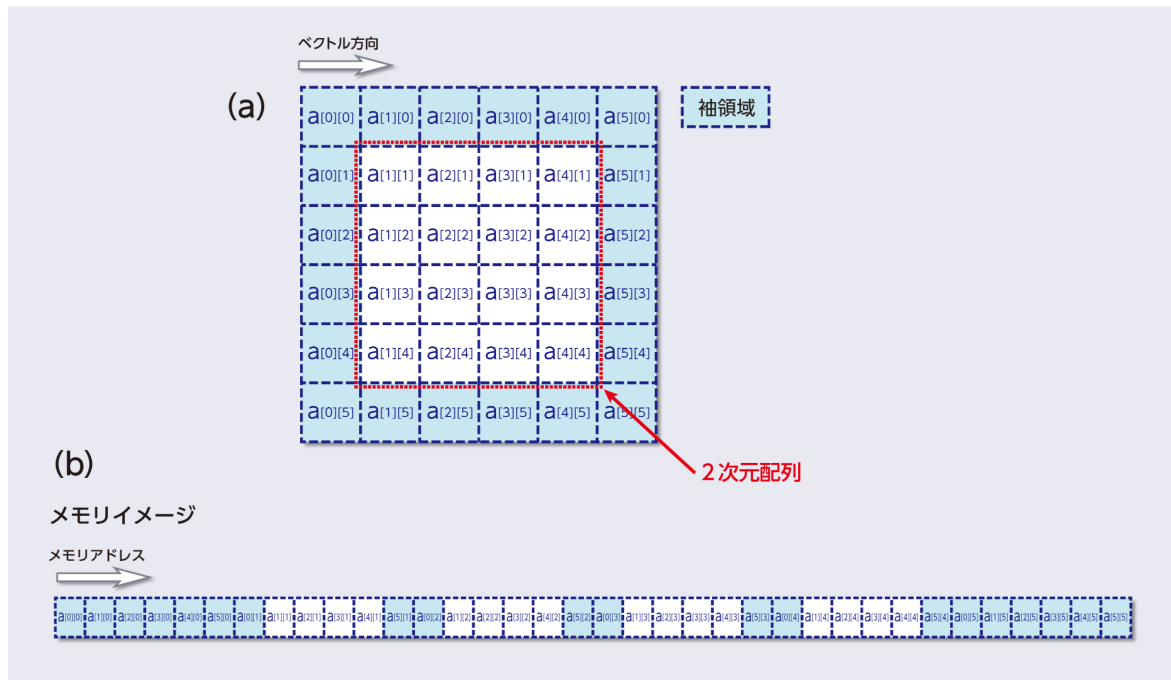


図 4.7　多次元配列と袖領域

```
      a[i][j] = ...;
    }
  }
```

ただし、第 5 章でも説明するとおり、リスト 4.28 の例のようにループ交換を行うと、メモリアクセスパターンが変化する場合があります。変化前のメモリアクセスパターンよりも変化後のほうがメモリアクセス性能を出しづらい場合、ループ交換によってベクトル長を増大させても実効性能はかえって低下してしまう恐れがあります。このように、最適化技法によっては副作用を持つものもあり、実際の高性能計算プログラミングにおいては適切なプログラムの書き方を試行錯誤によって探ることもしばしば必要になります。

第 4 章のポイント

- SX-Aurora TSUBASA で高い性能を達成するためには、コンパイラがベクトル化できると判断できるように配慮してループを書く必要があります。コンパイラがプログラムをどのようにコンパイルしたのかを、編集リストで確認することができます。
- コンパイラがベクトル化できないと判断した場合、その要因をベクトル化阻害要因といいます。実行に長時間を要するループがベクトル化されない場合には、依存関係などのベクトル化阻害要因を特定し、それがなくなるようにプログラムを修正する必要があります。
- ループの長さを増加させることにより、ベクトル化の効果が顕著になります。ただし、メモリアクセス性能とトレードオフがある場合には試行錯誤で適切な書き方を探る必要があります。

第 5 章

メモリアクセス最適化

SX-Aurora TSUBASA の VE の最大の長所は、非常に高い**理論メモリバンド幅**を持っていることです。その長所を活かすためには、とにかくプログラムがベクトル化されていることが大前提です。それに加えて、図 1.7 に示したような VE の**メモリシステム**の構造を考えると、メモリへのアクセスの仕方 (**メモリアクセスパターン**) によって**実効メモリバンド幅** (実際に達成されるメモリバンド幅) は大きく変化します [3]。本章では、VE のメモリシステムをより詳細に説明します。マイクロベンチマークを用いて様々なメモリアクセスパターンにおける実効メモリバンド幅を計測し、メモリアクセスを意識してループ構造やデータ構造を最適化する技法を説明します。実際のアプリケーションのプログラム最適化においては、メモリアクセス最適化をケースバイケースで考えていく必要がありますが、本章ではそれを検討するために必要なメモリアクセス最適化の基本概念や留意点を紹介します。

5.1 メモリシステムの構成

5.1.1 メインメモリ

VE には、第 2 世代の *High Bandwidth Memory* (*HBM2*) という規格の**メモリモジュール** (memory module) が使われています。図 5.1 に示すとおり、VE には合計で 6 つのメモリモジュールが接続されています。同図に示すとおり、1 つの HBM2 メモリモジュールは 8 つの**メモリチャネル** (memory channel) から構成されており、それぞれのメモリチャネルは独立に動作できます。各メモリチャネルは 128 ビットのインタフェースを持っていて、VE の場合には 1.6GHz で動作しています。このため、理論上は 128 bits×8 channels ×1.6 GHz=204.8 GB/s のバンド幅でデータを読み書きできます。したがって、VE に接続されている 6 つのモジュールのすべてのチャネルに並列にアクセスする理想的な状況を考えれば、LLC とメインメモリとの間の理論バンド幅は 1228.8 GB/s になることが分かります。実際には、様々なオーバヘッド (DRAM のリフレッシュ等) があるので理論バンド幅どおりの実効バンド幅を達成するのは不可能ですが、同世代のプロセッサの中では突出して高いメモリバンド幅を有しています。

HBM2 は、内部的には 128 バイトの**メモリセル**という単位でデータを保持しています。そして、メモリセルに記録されている 128 バイトのデータを一つの塊として入出力を行います。つまり、例えばプログラム中ではたった 1 バイトのデータにしかアクセスしていなかったとしても、内部的にはメモリからキャッシュ (*Last-level cache*, *LLC*) へ 128 バイトのデータが送られています。このため、メモリアドレスの連続した 128 バイトのデータをすべて使わない限り、メモリと LLC との間には無駄なデータ転送が発生します。また後述のように、異なるメモリセルにデータを分散配置して、それらを同時に読み書きすることで高いメモリバンド幅が実現されていますので、実際に同時アクセスされるメモリセルの数に依存して実効メモリバンド幅が変化します。このような理由から、メモリ上のデータへのアクセスの仕方によって実効メモリバンド幅は大きく変化するのです。

メモリと LLC とのインタフェースはチャネルあたり$\underline{128ビット}$ですので、1 つのメモリセル ($\underline{128バイト}$) の入出力に

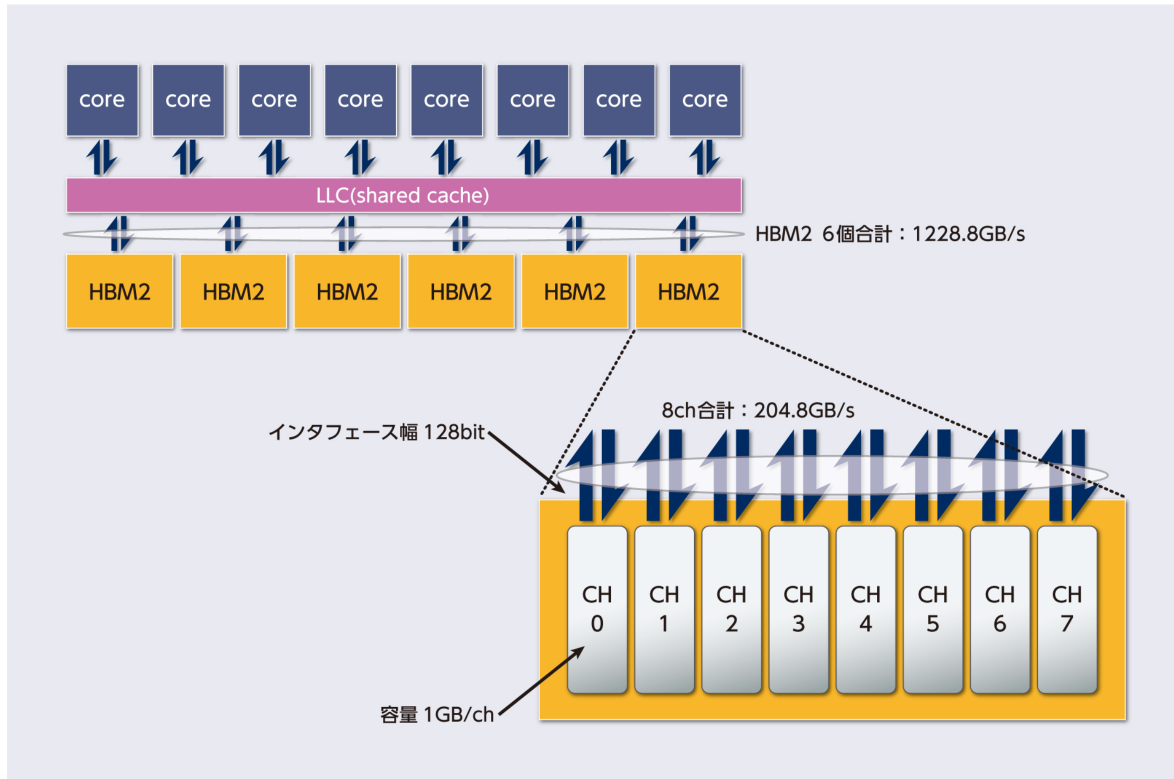


図 5.1　VE のメモリシステムの概要

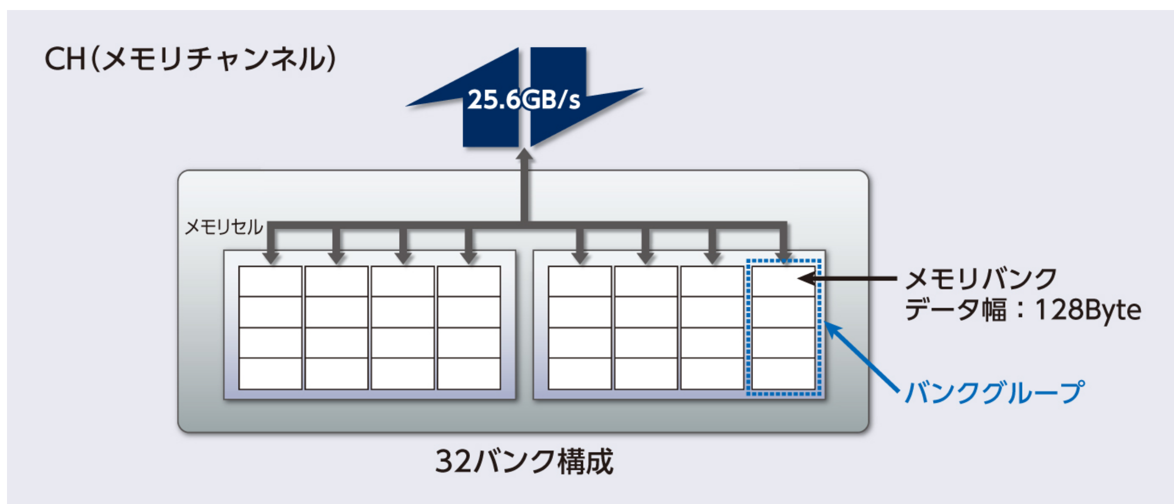


図 5.2　HBM2 のバンクの構成

は最短でも 8 サイクルかかります。実際には、メモリセルにアクセスする順番によって遅延時間が大きく異なります。図 5.2 に示すとおり、HBM2 の各チャネルを構成するメモリセルは、32 個の**メモリバンク**に分けられています。8 つのメモリバンクをまとめたものを、**メモリバンクグループ** (memory bank group) と呼びます。あるメモリセルにアクセスした後に、それとは異なるメモリバンクグループのメモリセルにアクセスする場合、最も短い遅延でアクセスできます。一方、同じメモリバンク内のメモリセルにアクセスする場合は、**バンク競合** (バンクコンフリクト) と呼ばれ、最も遅延時間が長くなります。同じメモリバンクグループ内でも、異なるメモリバンクのメモリセルであれば、バンク競合するよりは短い遅延時間でアクセスできます。アクセスの順番に依存して生じる遅延とは別に、データの読み込み (load) と書き出し (store) の切り替え時にも遅延が生じます。

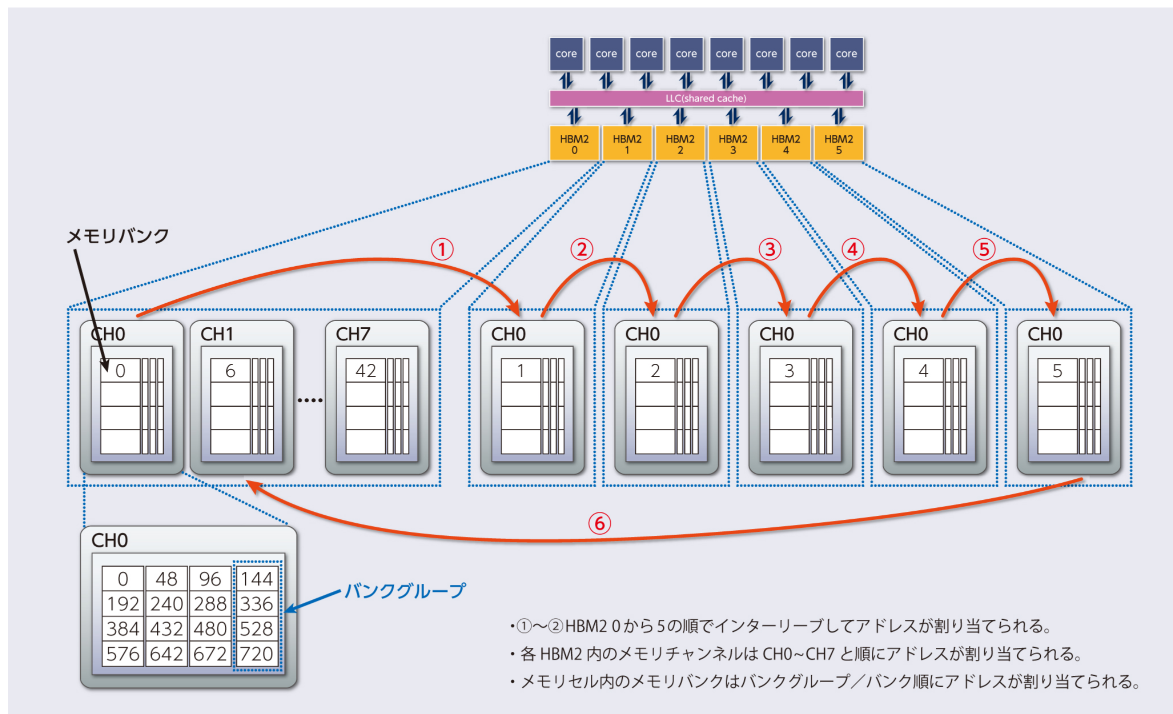


図 5.3　VE のメモリアドレス割当て

VE のメモリシステムは、メモリアドレス空間で隣接しているデータをなるべく別のモジュールやチャネルのメモリセルに割り当てることにより、メモリアドレス空間内で連続しているデータにアクセスする際に、複数のモジュールやチャネルに同時アクセスできるようになっています。メモリアドレス空間は 128 バイトごとに分割され、それぞれの別のモジュール/チャネル/バンクグループ/バンクのメモリセルに順番に割り当てられています。メモリアドレスとデータの配置との関係を、図 5.3 に示します。図中の①〜⑥のように、まずは HBM2 のモジュール 0〜5 の順でインターリーブしてアドレスが割り当てられます。次に、各 HBM2 モジュール内のメモリチャネルで、CH0 から CH7 と順にアドレスが割り当てられます。メモリモジュール内では、バンクグループ/バンクの順番にアドレスが割り当てられます。ここで、i 番目のモジュールの j 番目のチャネルの k 番目のバンクを $B(i, j, k)$ と書くことにしましょう。あるデータの最初の 128 バイト (1024 ビット) が $B(0, 0, 0)$ に対応づけられているとすると、次の 128 バイトは $B(1, 0, 0)$、その次の 128 バイトは $B(2, 0, 0)$ に対応づけられています。このように異なるモジュールに順番に対応付けていき、$B(5, 0, 0)$ の次には $B(0, 1, 0)$、すなわち 1 番目のモジュールの 2 番目のチャネルに対応づけられます。その後、各モジュールの 2 番めのチャネルに順番に対応付けていき、$B(5, 7, 0)$ の次は $B(0, 0, 1)$、すなわち 1 番目のモジュールの 1 番目のチャネルの別のメモリバンクに対応付けられます。

上述の構成から、以下のようなことがわかります。

- メモリアドレスの連続する 128 バイトのデータをすべて使わない限り、メモリと LLC との間のデータ転送が一部無駄になります。(VE に限らず) キャッシュを搭載する現代のプロセッサ全般に言えることですが、空間的局所性が高くなるようにデータ構造を考えたほうが、メモリとキャッシュ間の無駄な転送を避けるという観点で有利です。
- できるだけ多くのバンク/バンクグループ/チャネル/モジュールに含まれるメモリセルに同時にアクセスすることで、高い実効メモリバンド幅の達成を期待できます。ベクトルレジスタは 8 バイトの倍精度浮動小数点データを 256 要素まで保持できますので、1 回のベクトルロード/ストア命令で、最大で 8 bytes×256 elements=2048 bytes の連続するデータを読み書きすることになります。各メモリセルのデータは 128 バイトですので、1 回のベクトルロード/ストア命令実行では最大でもたった 16 個のメモリセルにしかアクセスしていません。それらのメモリセルは別のチャネルに含まれていますが、メモリと LLC との間のデータの通信路として 48 個のチャネルが用意されているにも関わらず、16 個 (すなわち全体の 1/3) のチャネルしか利用していないことになります。より多くのチャネルを使ってメモリアクセスするためには、
  - ループ長を長くして、ベクトルロード/ストア命令が連続的に実行されるようにする
  - 複数のコアから同時にメモリアクセスする

  などの方針が考えられます。
- 同じメモリバンク内のメモリセルに短期間内に集中するようなアクセスはバンク競合と呼ばれ、メモリアクセス性能が大きく低下するために避けたほうがよいといえます。具体的には、メモリアドレスで考えると 48 channels×32 banks×128 bytes=196608 bytes=192 KB だけ離れたデータが同じメモリバンク内のメモリセルに対応付けられていますので、短期間にアクセスするのは避けるべきです。

5.1.2 キャッシュ

VE には 8 モジュール構成の総容量 16MB の LLC が搭載されており、それを全てのコア (8 コア) で共有しています。VE のキャッシュ構成を図 5.4 に示します。再利用性の高いデータにアクセスするプログラムを最適化する際には、この LLC の有効利用も重要です。

コア側には 16 個のポート (ポートあたりのビット幅は 128) が用意されていて、キャッシュとのデータを送受信します。このため、各コアと LLC との間のバンド幅は 128 bits × 1.6 GHz × 16 ports = 409.6 GB/s になります。コアと LLC との間のネットワークは全体で 3072 GB/s のバンド幅を持っています。すなわち、8 コアすべてが LLC に同時アクセスしても十分なバンド幅を有していると言えます。ただし、各コアと LLC との間のバンド幅が 409.2 GB/s ということは、1 つのコアだけではメモリやキャッシュとの間のバンド幅を使いきれないということも分かります。バンド幅を有効活用するためには、複数のコアでメモリや LLC にアクセスするように、OpenMP などでプログラムを並列化する必要があります。並列化の効果については、第 5.2.1 節で紹介します。

LLC のラインサイズは 128 バイトです。すなわち、HBM2 のメモリセルのサイズと同じサイズになっています。メモリと同様、キャッシュでもアドレスの連続したデータが異なるモジュールのキャッシュラインに保持されます。CPU 側のポートと各モジュールとの対応関係が決まっているので、同じモジュールに保持されたキャッシュラインに同時にアクセスするようなプログラムでは、CPU ポートで衝突が発生します。第 3.3 節で説明した通り、FTRACE で CPU ポート衝突によって費やした時間を知ることができますので、衝突が頻発している場合には第 5.2.2 節で後述するようにメモリアクセスパターンを見直すことで性能が改善される可能性があります。

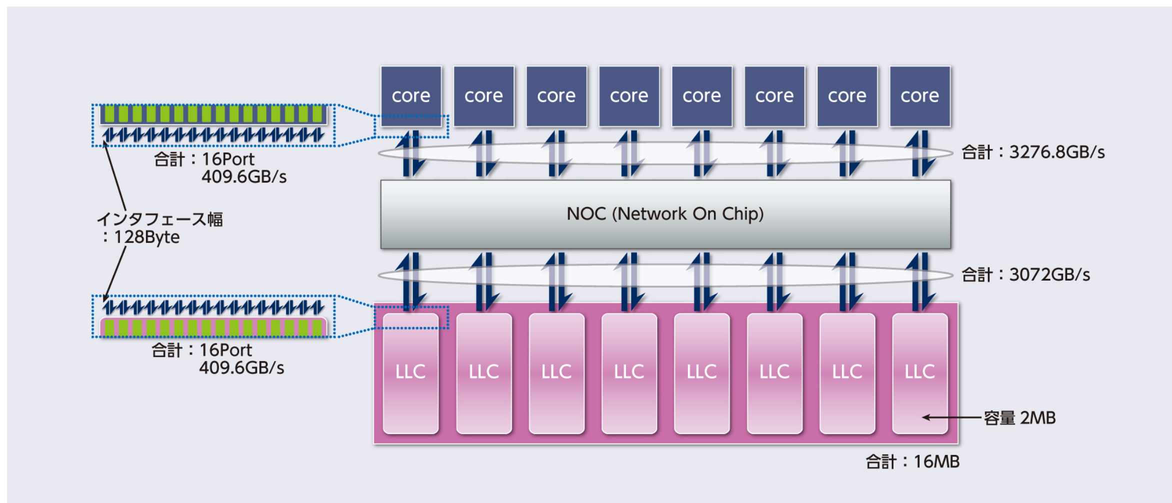


図 5.4　コアとキャッシュとの接続

5.2　メモリアクセス最適化

5.2.1　スレッド並列化によるメモリアクセス最適化

有名なルーフラインモデル [8] でも示されるように、多くの科学技術計算ではメモリ上にある大容量データに対する処理が求められ、**実効性能** (実際に達成される性能) の上限が実効メモリバンド幅によって決まります[*1]。そのようなプログラムでは、VE の持つメモリバンド幅を最大限に活用し、高い実効メモリバンド幅を達成することが、高い実効性能を達成するために必要不可欠です。

STREAM ベンチマーク [4] は、実効メモリバンド幅を計測する際に標準的に使用されるベンチマークです。その STREAM ベンチマークに含まれるプログラムを実行するとき、スレッド数 (すなわち、使用するコア数) の増加に伴う実効メモリバンド幅の変化を図 5.5 に示します。

STREAM ベンチマークには 4 種類のプログラム (Copy, Scale, Add, Triad) が含まれていますが、どのプログラムでも同様の結果が得られています。LLC と各コアとの間のバンド幅は 409.2 GB/s なので、1 つのコアだけでプログラムを実行している場合には、実効メモリバンド幅が 409.2 GB/s を超えることはできません。各コアで実行されているスレッドが常にメモリにアクセスするような処理を実行している場合、少なくとも 3 つのコアを使わないと VE の持つメモリバンド幅を最大限に利用することはできません。このため、OpenMP [2] 等を使ってプログラムをマルチスレッド化し、複数のコアを使ってプログラムを実行することは、マルチコアプロセッサである VE の演算性能を有効活用するだけではなく、メモリバンド幅を有効活用するためにも非常に重要です。

LLC とコアとの間にあるネットワークの総バンド幅は 3072 GB/s なので、再利用性が高く、常に LLC 上に必要なデータがあるような場合には 3072 GB/s のバンド幅でデータにアクセスすることを目指すことになります。この目標に近づくためには、コアあたりのバンド幅が 409.2 GB/s であることを考えると、8 コアすべてで効率よく LLC にアクセスしなければなりません。

以上のように、メインメモリとの間の実効メモリバンド幅、および LLC との実効バンド幅の両方の観点から、VE の 8 つのコアをすべて使って効率よくデータにアクセスすることが重要であることがわかります。並列処理を意識しな

*1 メモリバンド幅によって律速されると表現されます。

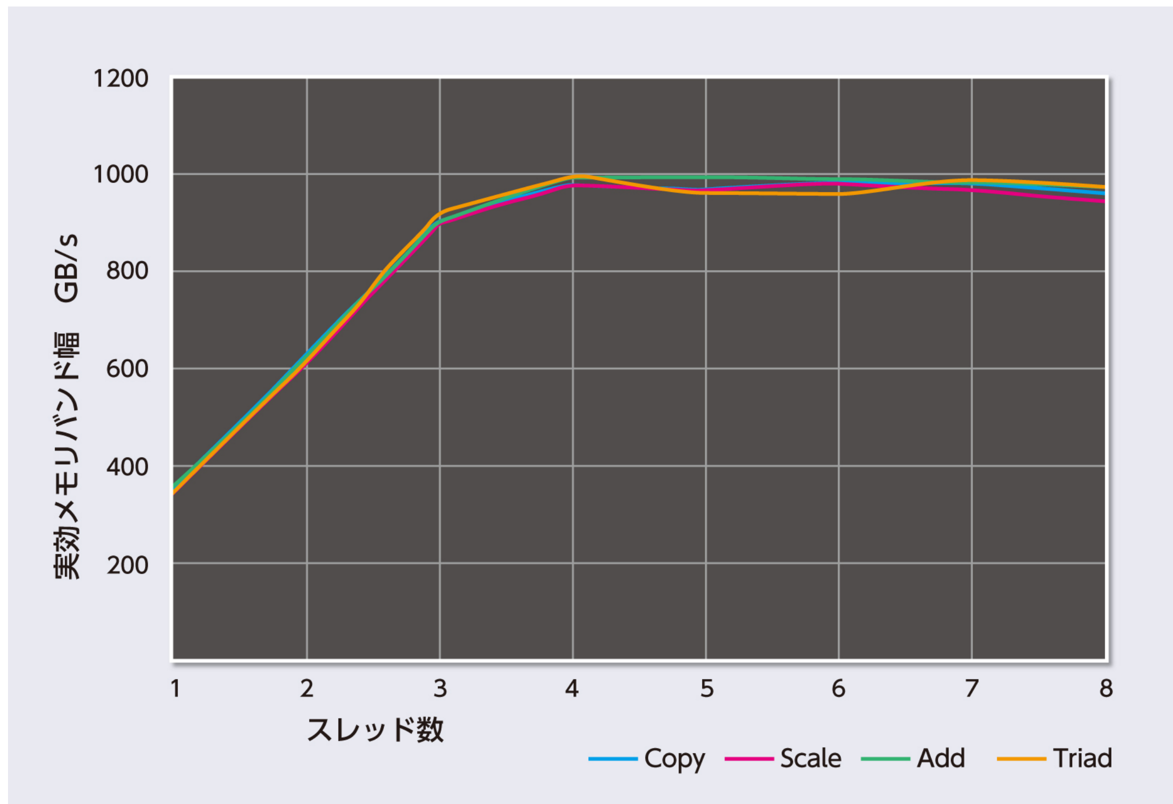


図 5.5　スレッド数と実効メモリバンド幅との関係

がら次節で述べるメモリアクセスパターンを最適化するのは容易ではありませんが、プロファイル情報*2で現状を観察しながらケースバイケースの対応をしていくことになります。

5.2.2　メモリアクセスパターン最適化

VE のメモリシステムは非常に高い理論メモリバンド幅を有していますが、任意のメモリアクセスパターンでその性能を期待できるわけではありません。メモリシステムの構造を考えると、高い実効メモリバンド幅を達成できるパターンはむしろ限られています。このため、高いメモリバンド幅を期待できるアクセスパターンにできるだけ近づけるようにループ構造やデータ構造を変更することにより、実際に実効メモリバンド幅を向上させ、その結果としてプログラムの実効性能を高めることができます。

典型的なメモリアクセスパターンとして、**逐次アクセス (シーケンシャルアクセス)**、**ストライドアクセス**、**リストアクセス**等があります。

逐次アクセスパターンとは、メモリアドレスの連続するメモリ上のデータ (配列要素等) に順番にアクセスしていくパターンです。

```
for(i=0;i<N;i++)
  a[i] = a[i] + b[i];
```

第 5.1 節で説明したとおり、VE のメモリシステム構成はこのシーケンシャルアクセスで最も高い実効メモリバンド幅を達成できるように設計されています。図 5.5 で使用されている STREAM ベンチマークは、シーケンシャルアクセス

*2 CPU ポート衝突、L1 キャッシュミス率、他には LLC？

パターンにおける実効メモリバンド幅を計測するプログラムです。このため、３スレッド以上で実行する場合には、理論メモリバンド幅に近い実効メモリバンド幅を達成できます。

ストライドアクセスパターンとは、決まった数 (**ストライド幅**) だけ要素を飛ばしてアクセスするパターンです。

```
for(i=0;i<N;i+=S)
  a[i] = a[i] + b[i];
```

この例では、ストライド幅 S だけ要素を飛ばして配列にアクセスしています。

また、構造体の配列 (Array of Structure, AoS) の場合、配列にシーケンシャルにアクセスしても、実際のアクセスとしてはストライドアクセスになります。

```
typedef struct {
  double x;
  double y;
} point;

point a[N],b[N];

for(i=0;i<N;i++)
  a[i].x = a[i].x + b[i].x;
```

この例の場合、構造体 point のメンバー変数である x と y はメモリ中に交互に並んでいますので (図 5.6)、各配列要素の x だけに順番にアクセスすると y の分だけ離れたアドレスにアクセスすることになります。このため、ストライド幅が 2 のストライドアクセスパターンになります。

構造体 pointの配列要素 1つ (a[0])

図 5.6　構造体の配列

図 5.7 に示す通り、ストライド幅が大きくなるとシーケンシャルアクセスとストライドアクセスとの性能差は広がります。これは、第 5.1 節で説明したメモリシステムの構成を考えると理解できます。HBM2 のメモリセルは 128 バイトであり、それを単位としてデータの読み書きを行います。例えば、倍精度浮動小数点データは 8 バイトなので、連続する 16 要素が 1 つのメモリセルに保存されています。このため、16 個の連続する配列要素にシーケンシャルアクセスする場合には、1 つのメモリセルだけにアクセスすれば良いことになります。一方、ストライド幅が S のストライドアクセスパターンで 16 個の配列要素にアクセスをする場合、必要なデータは S 個のメモリセルに分散していることになります (S が 16 より大きい場合には、16 個のメモリセルに分散している)。このため、16 個の配列要素にアクセスするために内部的には S 個のメモリセルにアクセスすることになり、S に反比例するように (見かけ上の) 実効メモリバンド幅が低下します[*3]。さらに、十分な長さのベクトルロード/ストア命令を多数のコアで実行する場合には、より多くのメモリセルに同時アクセスすることになりますので、同じメモリバンクやポートに同時にアクセスしてしまう確率が高まります。その結果として、実効メモリバンド幅がさらに低下する恐れがあります。

以上の理由から、図 5.7 でも見られるように、ストライド幅が大きくなるとシーケンシャルアクセスとストライドアクセスとの性能差が広がります。このため、ストライド幅がなるべく小さくなるように、できればシーケンシャルアク

[*3] 実際にメモリと LLC との間を流れるデータの量としては変わらないのですが、一部のデータしか使われないので無駄な転送となる。

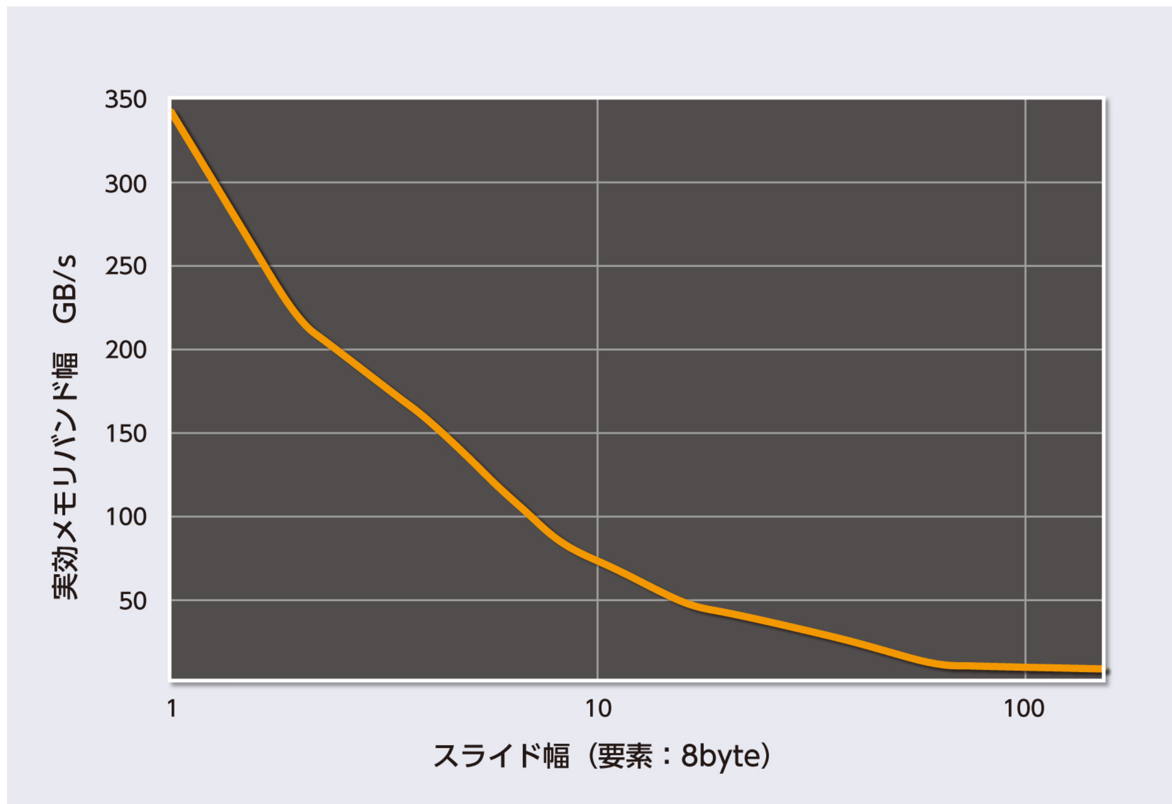


図 5.7　ストライド幅と実効メモリバンド幅との関係

セス (ストライド幅が 1 のストライドアクセスと考えることができます) になるようにプログラムを修正したほうが、実効メモリバンド幅の観点からは高い性能を期待できます。

リスト 4.28 で説明したループ交換はベクトル長を増やすためにも使われますが、ストライドアクセスパターンを回避したり、ストライド幅を小さくするためにもしばしば使われます。

Listing 5.1　ループ交換とストライド

```
 double a[N][M];

/* ループ交換前 */
for(i=0;i<N;i++){
  for(j=0;j<M;j++){
    a[j][i] = ...;
  }
}

/* ループ交換後 */
for(j=0;j<M;j++){
  for(i=0;i<N;i++){
    a[j][i] = ...;
  }
}
```

リスト 5.1 の例では、ループ交換をすることによってストライド幅が M のストライドアクセスをシーケンシャルアクセスに変えています。その結果として実効メモリバンド幅の改善を期待できます。ただし、もしも M が N よりも小さい

場合にはベクトル長が短くなってしまいますので、性能向上するとは限りません。

ループ構造を変化させるのではなく、メモリ内に並んでいるデータの順番を変更することでメモリアクセス性能が高くなる (なるべくストライド幅の小さいアクセスになる) ようにプログラムを最適化することを、**データレイアウト最適化**といいます。データレイアウト最適化の例をリスト 5.2 に示しています。配列 a と配列 b では次元の軸が入れ替わっている (配列要素が転置されている) ため、データレイアウト最適化後の配列へのアクセスはシーケンシャルになっています。

Listing 5.2　配列の転置によるデータレイアウト最適化

```
/* データレイアウト最適化前 */
double a[M][N];

for(i=0;i<N;i++){
  for(j=0;j<M;j++){
    a[j][i] = ...;
  }
}

/* データレイアウト最適化後  */
/* ただし a[i][j] == b[j][i] */
double b[N][M];

for(i=0;i<N;i++){
  for(j=0;j<M;j++){
    b[i][j] = ...;
  }
}
```

リスト 5.2 では 2 次元配列の例を示していますが、多次元配列の場合にも同様にして配列要素のメモリ上での並び順を変更するデータレイアウト最適化は効果的です。

また、構造体の配列 (Array of Structures, AoS) と比較して、配列の構造体 (Structure of Arrays, SoA) では配列要素がメモリ中に連続に並んでいるため、シーケンシャルアクセスパターンにしやすい傾向にあります (図 5.8)。このため、AoS を SoA に変換することも典型的なデータレイアウト最適化です。

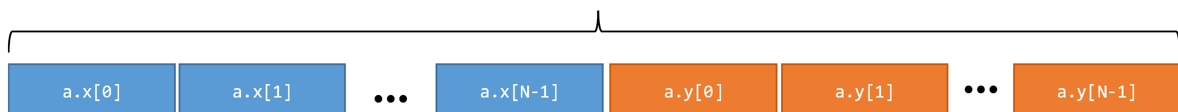


図 5.8　配列をメンバーに持つ構造体

Listing 5.3　AoS と SoA の変換によるデータレイアウト最適化

```
/* データレイアウト最適化前 */
typedef struct {
  double x;
  double y;
```

```
} point;

point a[N]; /* 構造体型変数の配列 */
for(i=0;i<N;i++){
  a[i].x = ...;
}

/* データレイアウト最適化後  */
typedef struct {
  double x[N];
  double y[N];
} point;

point b; /* 配列をメンバーに持つ構造体 */
for(i=0;i<N;i++){
  b.x[i] = ...;
}
```

リストアクセスパターンとは、ある配列の要素をインデックス (添字) として使って、別の配列の要素にアクセスするパターンです。

```
for(i=0;i<N;i++)
  a[b[i]] = ...;
```

この例では、配列 b の要素をインデックスとして使って、配列 a の要素にアクセスしています。配列 b のようにインデックスが保存された配列のことを、**リスト配列**と呼びます。リストアクセスパターンの場合には、リスト配列 (b) とデータ配列 (a) の 2 つの配列にアクセスしなければならず、しかもリスト配列の要素の値を読み込んでからデータ配列の要素にアクセスしなければなりません。また、リスト配列の要素の値によってデータ配列へのアクセスパターンが決まるので、多くの場合は配列へのアクセスがシーケンシャルにはならず、ストライドアクセスパターンと同様の理由で性能が低下します。

リストアクセスパターンの場合、そのメモリアクセス性能はリスト配列の内容に依存します。リスト配列にアクセスしてからデータ配列にアクセスしますので、リスト配列に高速にアクセスすることが望まれます。このように、高速にアクセスしなければならない配列が決まっている場合には、その配列をキャッシュに保持することで性能が向上します。SX-Aurora TSUBASA のコンパイラでは、以下の**コンパイラ指示行**でリスト配列 b をキャッシュ上におくことをコンパイラに指示することができ、配列 b へのアクセスを高速化することで配列 a へのリストアクセスが高速化するかもしれません。

```
#pragma _NEC retain(b)
for(i=0;i<N;i++)
  a[b[i]] = ...;
```

このコンパイラ指示行を有効にする場合にはコンパイラオプションとして`-mretain-list-vector` あるいは`-mretain-none` を指定する必要があるため、キャッシュにおくデータをコンパイラやシステムが検討する余地を制限することになり、結果として遅くなるかもしれません。ケースバイケースでの使い分けを考えましょう。

リストベクトルが長さ 256 以下の一次元ローカル配列のとき、いくつかの条件を満たしていればベクトルレジスタに保持することができるかもしれません。これをコンパイラに明示的に指示するために、以下のコンパイラ指示行を使うことができます。

```
#pragma _NEC vreg(b)
```

```
for(i=0;i<N;i++)
  a[b[i]] = ...;
```

この場合、関数内の配列 texttb へのアクセスはすべてベクトルレジスタへのアクセスになります。このため配列 b の要素に頻繁にアクセスする場合には処理の高速化を期待できます。

5.3　メモリ階層の効果的利用

第 5.2.2 節では、レジスタやキャッシュを利用することでリストアクセスパターンの実効メモリバンド幅を改善できる可能性を示しましたが、他にも様々な最適化技法においてレジスタやキャッシュが重要な役割を果たします。

レジスタやキャッシュを上手に利用することで、メモリアクセスの回数そのものを減らすこともできます。例えば、リスト 5.4 のオリジナルプログラムでは配列 a に 2*N*N*N 回アクセスしていることがわかります。すなわち、N*N*N 回の繰り返しのそれぞれで、配列 a の要素を 1 回だけ読み込んで、計算後に 1 回だけ書き戻します。しかし、**アウターアンローリング** (outer unrolling) を行うことで、メモリアクセス回数を N*N*N 回に削減できています。つまり、配列 a の要素を 1 回読み込んだら、ループ k の繰り返し 2 回分の計算を行い、その結果を配列 a の要素として書き戻しています。この例では短いプログラムなので気にならないかもしれませんが、一般的にプログラムが読みにくくなってしまいますので、同等の変換を行う指示行も用意されています。この例ではループ k に関して繰り返し 2 回分を計算しています。

Listing 5.4　ループ展開によるメモリアクセス回数削減

```
/* オリジナル */
for(j=0;j<N;j++)
  for(k=0;k<N;k++)
    for(i=0;i<N;i++)
      a[j*N+i]=a[j*N+i]+b[k*N+i]*c[j*N+k];

/* アウターアンローリング */
for(j=0;j<N;j++)
  for(k=0;k<N;k+=2)
    for(i=0;i<N;i++)
      a[j*N+i]=a[j*N+i]+b[k*N+i]*c[j*N+k]
                       +b[(k+1)*N+i]*c[j*N+k+1];

/* コンパイラ指示行による同等の変換 */
for(j=0;j<N;j++)
#pragma _NEC outerloop_unroll(2)
  for(k=0;k<N;k++)
    for(i=0;i<N;i++)
      a[j*N+i]=a[j*N+i]+b[k*N+i]*c[j*N+k];
```

第 5 章のポイント

- VE が達成できる実効メモリバンド幅は、メモリアクセスパターンに大きく依存します。メモリシステムの構造を考え、なるべく効率のよいメモリアクセスとなるようにプログラムを工夫することで、実効メモリバンド幅が大きく改善し、その結果として実効性能も大きく改善します。
- シーケンシャルアクセスパターンのときに、最も高い実効メモリバンド幅を達成できます。ストライドアクセス

の場合には、基本的にストライドが小さい方が高い実効メモリバンド幅を期待できます。ストライドを短くするための工夫の典型例として、ループ交換やデータレイアウト最適化が挙げられます。

- 再利用性が高いデータや、リスト配列のように高速にアクセスしなければならないデータがある場合、それらを優先的にキャッシュやレジスタに保持するように、コンパイラ指示行で明示的に指示することができます。ただし、その結果として実効メモリバンド幅や実効性能が向上するかいなかはケースバイケースです。

第Ⅱ部

応用編

第 6 章

異種プロセッサ間の連携

第 7 章

プログラム開発事例

第 8 章

その他の高度な話題

付録 A

OpenMP 超入門

第 1 章でも説明したとおり、VE は 8 コアを搭載するマルチコアプロセッサです。したがって、1 つのプログラムを複数のコアで手分けして実行することができれば、実行時間を短縮することができます。しかし、普通にプログラムを書いただけでは、1 つのコアしか使われません。SX-Aurora TSUBASA に限らず、現在の一般的な計算システムで高い性能を達成するためには、複数のコアや複数のコンピュータで処理を手分けして実行する、すなわち**並列処理**するように、明示的にプログラム中で指示する必要があります。そのようなプログラムの作成を**並列プログラミング**と呼びます。また、逐次処理のプログラムを並列処理されるように修正あるいは変換することを、プログラムの**並列化**と呼びます。VE の 8 つのコアは、同じメモリを共有している構成になっています。このため、同じデータにアクセスできる状態で最大 8 つのコアが並列処理を行うことができます。このようにメモリを共有して行う並列処理を、**共有メモリ型並列処理**と呼びます。

Linux などの一般的な OS の環境でプログラムを実行すると、その実行のために必要な計算資源 (**リソース**) が確保されて**プロセス**という単位でまとめて管理されます。例えば、リスト 2.1 のように関数 `malloc` でメモリ領域を確保すると、確保されたメモリ領域はプロセスに対して割り当てられます。あるプロセスに割り当てられたリソースに対して、別のプロセスからアクセスすることはできません。このため、VE のようなマルチコアプロセッサにおける共有メモリ型並列処理では、その名の通り同じメモリを複数のコアで共有しながら並列に処理を実行するために、各コアにおける処理が (プロセスではなく) **スレッド**という単位で通常管理されます。1 つのプロセスで複数のスレッドを生成し、そのプロセスに割り当てられたリソースをスレッド間で共有しながら、複数のコアのそれぞれでスレッドを実行することになります。スレッド間では様々なリソースを共有することもできますし、私有のリソースを確保することもできます。プログラムを複数のスレッドで実行することを**マルチスレッド実行**と呼び、マルチスレッド実行されるようにプログラムを修正することを**マルチスレッド化**と呼びます。

第 A 章では、共有メモリ型並列処理のための標準プログラミング基盤である *OpenMP* [2] について、ごく初歩的な使い方を説明します。プログラムのどの部分を複数のコアで実行し、どのデータをスレッド間で共有するか、どのデータを各スレッドの私有リソースとして割り当てるか、などを指示する方法を述べます。

OpenMP は共有メモリ型並列処理を記述するために用いられる標準プログラミング基盤です。この OpenMP の使い方を詳しく紹介するためには、それだけで本が一冊かけてしまうくらいの様々な説明が必要になりますが、基本的な使い方だけであれば意外と手軽に使えます。以下では、OpenMP の最も基本的な使い方として、1 つのループを複数の**スレッド**で手分けして実行 (**ワークシェアリング**) する方法を紹介します。基本的で簡単な使い方ではありますが、最もよく出てくる OpenMP の用途だといえます。

第 2 章のリスト 2.1 に示したプログラムを OpenMP で並列化した例を、リスト A.1 に示します。

Listing A.1　OpenMP による行列積プログラムの並列化

```
1  #include <stdio.h>
2  #include <stdlib.h>
```

```
3  #define N (2048)
4
5  int main(int argc, char** argv) {
6    int i,j,k;
7    double *a, *b, *c;
8
9    a = (double*)malloc(N*N*sizeof(double));
10   b = (double*)malloc(N*N*sizeof(double));
11   c = (double*)malloc(N*N*sizeof(double));
12
13   /* 行列の初期化 */
14   for(j=0;j<N;j++){
15     for(i=0;i<N;i++){
16       a[j*N+i]=i*j;
17       b[j*N+i]=(i==j?1:0);
18       c[j*N+i]=0.0;
19     }
20   }
21
22   /* 行列積の計算 */
23 #pragma omp parallel for private(i,j)
24   for(k=0;k<N;k++){
25     for(j=0;j<N;j++){
26       c[k*N+j] = 0.0;
27       for(i=0;i<N;i++){
28         c[k*N+j] += a[k*N+i]*b[i*N+j];
29       }
30     }
31   }
32
33   return 0;
34 }
```

このプログラムはリスト 2.1 とほとんど変わりませんが、行列積を計算する三重ループの直前に`#pragma omp`で始まるコンパイラ指示行が追加されています。OpenMP では、このようなコンパイラ指示行によってプログラムをどのように並列実行するのかを指定します。リスト A.1 の例では、`#pragma omp parallel for`を挿入することによって指示行直後の`for`ループを複数のスレッドで並列実行 (ワークシェアリング) することを指示しています。すなわち、直後の`for`ループは`k`が`0`から`N-1`になるまで処理を`N`回繰り返しますが、その`N`回分の処理を複数のスレッドで手分けして実行することを指示しています。

このプログラムを`ncc`でコンパイルする場合、`-fopenmp`オプションをつけることでコンパイラは OpenMP の指示行に基づいてプログラムを並列化します。プログラムを実行するスレッドの数は、環境変数`OMP_NUM_THREADS`で指定します。以下の例では、VE 上で 2 スレッドで実行されます。

```
$ ncc -fopenmp matmul.c -o matmul
$ export OMP_NUM_THREADS=2
$ ve_exec ./matmul
```

リスト A.1 で`N`は 2048 なので、2 つのスレッドでワークシェアリングする場合には、`k`が`0`から`1023`までは 1 つ

目のスレッド、1024 から 2047 までは 2 つ目のスレッドというふうに処理が均等に分担されます。

何も指示しなければ、ループ本体 ({ と } で囲まれた**ブロック**の内部 (25 行目から 30 行目まで)) で使われるすべてのデータはスレッド間で共有されています。すなわち何も指示しなければ、変数 j や変数 i もスレッド間で共有されています。あるスレッドが j=0 を実行すると、他のスレッドにとっても変数 j の値は 0 になるということです。あるスレッドが j++ を実行すると、他のスレッドにとっても変数 j の値が 1 増えることになります。プログラムとしては内側の 2 つの for ループはそれぞれ N 回ずつ処理を繰り返すことが想定されていますが、変数 j や変数 i がスレッド間で共有された状態では、他のスレッドによってそれらの変数の値が勝手に変えられてしまうので期待通りの動作になりません。そこで変数 j や変数 i は各スレッドに固有の値をもつ**私有変数**であることを private(i,j) で指示しています。その結果、リスト A.1 は正常に行列積の計算を行うことができます。なお、変数 k は自動的に私有変数として扱われます。

変数がスレッド間で共有されたままだと困る典型例として、**リダクション変数**もあります。

Listing A.2 OpenMP による合計計算の並列化

```
1    int total=0;
2
3  #pragma omp parallel for reduction(+:total)
4    for(i=1;i<=100;i++){
5      total += i;
6    }
```

リスト A.2 で、変数 total に対して + 演算子による**リダクション演算**が行われているので、変数 total はリダクション変数です。もし仮に複数のスレッドが完全に同時に total+=i を実行したら、変数 total の値はどのようになるかわかりません。そのような状況を防ぐために、指示行中の reduction(+:total) によって、total が + 演算子によるリダクション変数であることが示されています。その結果として、リスト A.2 のループは 1 から 100 までの総和を正常に計算できます。

以上のように、変数の共有と私有に配慮しながらループを#pragma omp parallel for でワークシェアリングすることにより、比較的容易にループの繰り返し処理を複数コアで分担して実行することができます。また、リダクション演算はよく使われるので、リダクション演算を含むループを正常にワークシェアリングするための記述方法も用意されています。それ以外にも OpenMP は非常に多くの機能を提供しているので、そのすべてを本文書で紹介することはできませんが、上述の機能だけでも OpenMP のよる高速化の効果を確認できます。例えば、環境変数 OMP_NUM_THREADS の値を変化させながら、リスト A.1 のプログラムを実行すれば、複数スレッドで手分けをして並列処理することによる高速化の効果を見ることができます。この例のように大量の演算を実行しなければならない場合、それらを複数のコアで並列実行することができれば、実行時間を大幅に短縮できることは想像に難くないでしょう。VE は 8 コアを搭載するプロセッサですので、その演算性能を活用するためにはマルチスレッド化が必要です。VE の 1 コアのピーク性能は 307.2 Gflop/s であり、すべてのコアを使ったときのピーク性能は $8 \times 307.2 = 2457.6$ Gflop/s となります。

参考文献

[1] NEC SDK: C/C++ Compiler ユーザーズガイド (Japanese) Revision 8. https://www.hpc.nec/documents/sdk/, 2019.

[2] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2007.

[3] Naoki Ebata, Ryusuke Egawa, Yoko Isobe, Ryoji Takaki, and Hiroyuki Takizawa. Memory First : A performance tuning strategy focusing on memory access patterns, 2019.

[4] John D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. https://www.cs.virginia.edu/stream/.

[5] Peter S. Pacheco. MPI 並列プログラミング. 培風館, 2001. 秋葉 博 訳.

[6] David A. Patternson and John L. Hennessy. コンピュータの構成と設計　～ハードウェアとソフトウェアのインタフェース～. 日経 BP 社, 第 5 版 (上), 1996. 成田 光彰 訳.

[7] Takashi Soga, Akihiro Musa, Youichi Shimomura, Ryusuke Egawa, Ken'ichi Itakura, Hiroyuki Takizawa, Koki Okabe, and Hiroaki Kobayashi. Performance evaluation of NEC SX-9 using real science and engineering applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–12. IEEE, 2009.

[8] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, Vol. 52, No. 4, pp. 65–76, 2009.

[9] 塩月信智, 江川隆輔, 滝沢寛之. SX-Aurora TSUBASA におけるプロセス間通信の性能評価. 研究報告ハイパフォーマンスコンピューティング (HPC), Vol. 2018-HPC-165, No. 21, pp. 1–6, 2018.

索引

HBM2, 12
High Bandwidth Memory, 12, 63

Last-level cache, 63
LLC, 63

アウターアンローリング, 73
アウターアンロール, 48
アクセス遅延時間, 34
依存関係, 53
インダクション変数, 51, 55
インデックス, 56
InfiniBand, 21
インライン展開, 40, 52
HBM2, 63
FMA, 14
MPI, 19
AVX-512 命令, 10
OpenMP, 19, 89
拡散, 49
管理者権限, 22
カーネル, 15
カーネル空間, 15
キャッシュヒット, 34
キャッシュミス, 34
共有オブジェクト, 77
共有メモリ型並列処理, 89
ギャザー, 49
コア, 9
交差ループ長, 34
高性能計算プログラミング, 3
コンテキスト, 82
コンパイラ, 43
コンパイラ指示行, 56, 72, 90
コンパイル, 3
コンパイルリスト, 45
最内ループ, 50
システムコール, 15
SIMD 命令, 3
収集, 49
私有変数, 91
診断メッセージリスト, 45
シーケンシャルアクセス, 56, 68
CPU ポート衝突, 38
実効性能, 10, 67
実効メモリバンド幅, 13, 63
自動ベクトル化機能, 49
スカラプロセッサ, 10
スカラユニット, 14
スキャッタ, 49
ストライドアクセス, 56, 68
ストライド幅, 69
ストリップマイニング, 37
スループット, 44
スレッド, 83, 89
添字, 56
ソケット, 9
袖領域, 60
増分式, 51
増分値, 51
立ち上がり時間, 49
遅延時間, 43
逐次アクセス, 68
逐次実行, 43
デバイス ID, 19
デバッガ, 27
データレイアウト最適化, 71
動的リンク, 77
バンク競合, 65
バンクコンフリクト, 65
パイプライン実行, 44
プロセス, 15, 82, 89
並行実行, 58
並列化, 89
並列処理, 89
並列プログラミング, 89
編集リスト, 45
ベクトルアーキテクチャ, 3
ベクトル演算命令, 13
ベクトル演算器, 44
ベクトル演算率, 33
Vector Engine Offloading, 4, 77
ベクトル化, 3, 43
ベクトル化阻害要因, 49
ベクトル化率, 33
ベクトル型計算システム, 3
ベクトルプログラミング, 3, 43
Vector Host Call, 4, 77
ベクトル命令, 3
ベクトルレジスタ, 13
ベクトルロード/ストア命令, 13
マクロ演算, 50, 57
マルチスレッド化, 89
マルチスレッド実行, 89
メモリアクセスパターン, 63
メモリシステム, 63
メモリセル, 63
メモリバンク, 65
メモリバンド幅, 10
メモリモジュール, 63
融合積和演算, 14
ユーザ空間, 15
ラストレベルキャッシュ, 35
リストアクセス, 56, 68
リスト配列, 72
リソース, 15, 82, 89
リダクション演算, 91
リダクション変数, 91
理論演算性能, 9
理論メモリバンド幅, 10, 63
root 権限, 22
ループ一重化, 59
ループ交換, 50, 60
ループ制御変数, 51
ループ長, 58
ループ展開, 50
ループ内不変式, 51
ループネスト, 50

ループ融合, 60
レイテンシ, 34, 43
レベル 1 キャッシュ, 35
ワークシェアリング, 89